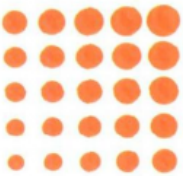
大规模软件构架技术

王映辉　冯德民　编著

科学出版社
www.sciencep.com

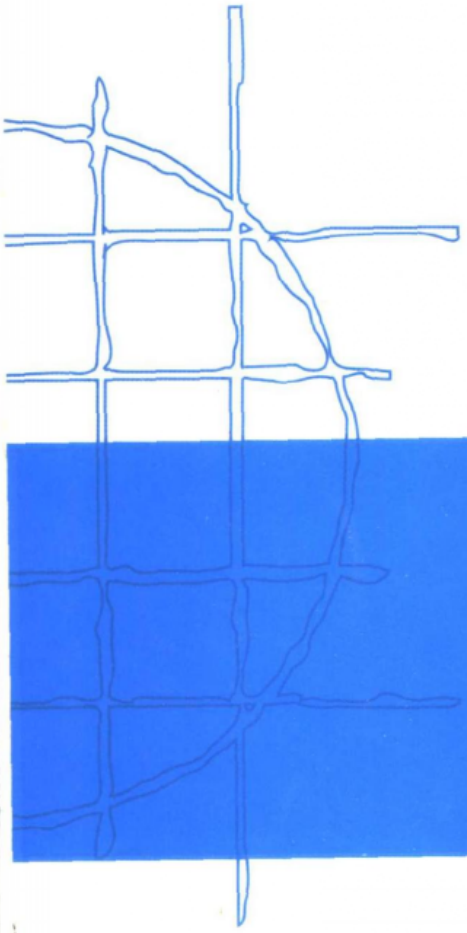
(TP-1984.0101)

责任编辑：赵卫江



本书主要内容：

- 面向对象技术简介
- 分布式处理技术的内涵
- 开放式分布处理ODP的参考模型和体系结构
- 中间件技术和分布构件模型技术
- 软件Agent和MAS技术
- 各种构件模型的集成方法和技术
- 应用实例：数字城市的软件构架模型



ISBN 7-03-011517-1

9 787030 115171 >

科学出版社 技术分社
http://www.abook.cn

ISBN 7-03-011517-1
定 价：25.00 元

陕西师范大学出版基金资助出版

大规模软件构架技术

王映辉　冯德民　编著

科学出版社

北　京

内 容 简 介

大规模软件构架技术是近几年发展起来的一个重要分支学科。本书比较全面地描述了大规模软件构架的关键技术，揭示了大规模软件构架的内涵。本书共7章。第1章简要总结了面向对象技术；第2章给出了分布式处理技术的内涵、开放式分布处理ODP的参考模型和体系结构；第3，4章描述了中间件技术和该技术支持下的几种分布构件模型技术；第5章阐述了软件Agent和MAS技术；第6章总结了各种构件模型的集成方法和技术；第7章给出了基于大规模软件构架技术的应用实例，即数字城市的软件构架模型

本书可作为构建分布式环境系统、企业电子商务平台系统，特别是空间信息平台软件的科研人员、高等院校教师和研究生的参考书。

图书在版编目(CIP)数据

大规模软件构架技术/王映辉等编著. —北京：科学出版社，2003

ISBN 7-03-011517-1

Ⅰ.大… Ⅱ.王… Ⅲ.软件开发 Ⅳ.TP311.52

中国版本图书馆CIP数据核字（2003）第038324号

责任编辑：赵卫江/责任校对：郝 岚

责任印制：吕春珉/封面设计：一克米工作室

科学出版社 出版

北京东黄城根北街16号

邮政编码：100717

http://www.sciencep.com

新蕾印刷厂 印刷

科学出版社发行 各地新华书店经销

2003年6月第 一 版 开本：B5（720×1000）

2003年6月第一次印刷 印张：13

印数：1—4 000 字数：240 000

定价：25.00元

（如有印装质量问题，我社负责调换〈路通〉）

作者简介

王映辉，分别于1989年、1999年和2002年毕业于陕西师范大学（本科）、西南石油学院（硕士）和西北大学（博士），同时分别获得计算机软件学士、矿产普查与勘探硕士和计算机软件与理论博士学位。现为陕西师范大学计算机科学学院副教授，从事计算机软件与理论教学与科研工作。曾有10多年的石油工程软件开发经验，并荣获厅局级科技进步奖多项。近几年作为主要完成人员参加过863高科技攻关项目、国家自然科学基金项目、陕西省科学基金项目和西安市科技攻关项目等多项研究工作。近三年内在核心及以上级别的期刊上发表论文近20篇。目前的主要研究方向是大规模软件技术、信息可视化技术与WebGIS。

冯德民，教授。长期从事计算机软件与理论教学与科研工作，先后主持或作为主要参加人员完成了多项国家级和省部级科研项目。在核心及以上级别的期刊上发表论文20多篇。现为陕西师范大学计算机科学学院院长，主要研究方向为软件工程和算法理论。

前　言

计算机软件的发展从面向过程到面向对象，直到面向软件 Agent，无论从开发方法上，还是从开发工具上都发生了翻天覆地的变化，从而使人们的开发模式从简单的单机模式发展到了复杂的分布式大规模软件集成模式。

1. 计算模式的发展

在计算技术领域，随着微处理器技术和计算机网络技术的不断发展，计算模式经过了以下几次变迁。

(1) 集中式计算模式

从 1945 年现代计算机时代开始到 1985 年前后，计算机设备既庞大又昂贵，即使是小型机，每台也价值数万美元。因此，大多数机构也只有有限的几台计算机。为了节省成本，在一个系统中往往以一台主机（mainframe）为主，连接着若干个终端设备。所有的数据存储和计算都在主机上进行，终端设备只负责为用户发出计算请求和显示计算结果。我们称这种方式为集中式计算模式。

(2) 桌面计算模式

随着集成电路技术的不断进步，20 世纪 80 年代中期开始出现了微型计算机。从 8 位、16 位、32 位，到今天 64 位的 CPU 机型，发展非常迅速。许多 PC 机和工作站具备了以前大型计算机的能力，可以存储大量的数据且能进行相对复杂的计算，而价格却非常便宜，可以被一个机构大量采用。计算机也由此脱下了高贵的外衣，走入了寻常百姓之家。因此计算模式的主流从主机转移到了用户桌面。我们称这个阶段的模式为桌面计算模式。

(3) 分布式计算模式

进入 20 世纪 90 年代，计算机技术最显著的进步之一就是高速计算机网络技术的飞速发展。局域网 LAN 使得同一建筑内的数十甚至上百台计算机连接起来，使大量的信息能够以 $10^8 \sim 10^9$bps 或更高的速度在计算机间传送。广域网 WAN，尤其是 Internet 的迅速普及，使得全球范围内的数百万台计算机连接起来得以进行信息交换，改变了人们传统的获取和处理信息的方式。随着计算资源的网络化，拥有个人计算机或工作站的广大用户，迫切需要共享分布于网络上的丰富信息资源，用以廉价获得超出局部计算机能力的高品质服务，并逐步实现具有计算机支持的协同工作，因此在多个资源上进行分布式处理就变得越来越迫切。从简单的数据共享到多个服务的先进系统，大量的计算转移到了网络环境下

的各种资源和个人桌面，这就是分布式计算模式。分布式计算技术成为影响当今计算机技术发展的关键技术。

2. 分布式计算模式的特征

分布式计算（distributed computing）技术是在近 20 年来影响计算技术发展的最为活跃的因素之一，它的发展经历了两种不同的技术路线：理想路线和现实路线。

（1）理想路线（传统意义上的理解）

对于分布式计算或者称分布式系统、分布式应用，不同的学者给出了不同的定义。A.S.Tanenbaum 认为一个分布式系统可以看作是一些独立的计算机集合，但是对这个系统的用户来说，系统就像一台计算机一样。这个定义有两个方面的含义：第一，从硬件角度来讲，每台计算机都是自主的；第二，从软件角度来讲，用户将整个系统看作是一台计算机。Carl L.Hall 则将分布式计算定义为通过多个独立的计算机处理来完成一个特定的任务，每个处理可以在相同或不同的计算机平台上以并行或串行的方式进行，通过通信协议相互协作完成任务，从而实现了把计算负担分散到多个能通信的计算机上。对于分布式应用系统，P.H.Enslow 作了如下描述：包含许多物理资源和逻辑资源；通过网络通信实现信息交换；有一个高层的操作系统能够对整个系统进行管理；系统对用户透明；系统中各部分资源既相互独立又相互配合。

从上面的定义不难看出，系统中计算机的互连和各部分在网络中的分布仅仅是分布式计算的必要条件，分布式系统的统一的逻辑特性才是其充分条件，主要特征有以下几点：

①透明性。就是让用户将一些机器集合的协同工作看作是在一台机器上完成的。其主要包括：位置透明，即用户不需知道软、硬件资源（如 CPU、文件和数据库）的位置，资源的名字不应含有资源的位置信息；迁移透明，即资源无需更名就可自由地在系统中流动，外界不需要知道系统为使资源均衡而改变对象的位置；复制透明，即系统可以随意地为文件和其他资源进行附加拷贝而无需用户知道；并发透明，即多个用户可以自动共享资源；并行透明，即系统活动可以在用户没有感觉的情况下并行发生；存取透明，即隐藏数据表示和调用机制的异同；失败透明，即将出错和恢复事件隐藏在对象内部，以达到容错的目的；持久性透明，即对象里隐藏着所用资源的变化，如处理器资源、存储资源的冻结与解冻；重定位透明，即改变一个接口的位置不影响与之编联的其他接口；提交透明，即一组对象发生作用的次序不影响结果的一致性等。

②灵活性。可以根据不同的情况，用最有效的方式将工作负荷分配到可用的机器上，最大限度地合理利用资源。

③可靠性。即系统可以屏蔽错误。通过把工作负载分散到众多的机器上，单个芯片的故障最多只会使一台机器停机，而其他机器不会受任何影响。理想条件下，某一时刻如果有5%的计算机出现故障，系统将仍能继续工作，只不过损失5%的性能。对于某些关键性应用，如证券交易或核反应堆控制系统，采用分布式系统可以保证其高可靠性。

④可伸缩性。系统可在需求增长的时候，通过增加资源，对系统能力进行灵活地扩充。

由此可见，理想路线试图在互连的计算机硬件上部署全新的分布式操作系统，全面管理系统中各自独立的计算机，呈现给用户单一的系统视图。在20世纪80年代，学术界普遍追求这一目标，尽管产生了许多技术成果和实验系统，但却没有被用户和市场接受。这些技术总称为分布式计算机系统技术。对于分布式计算机系统的定义一直不太明确的一个主要原因是，不知采用什么样的模型来构造、管理和协调使用基于网络（包括局域网、城域网和广域网）中的各个计算机系统。但是，对于分布式计算机系统，当时追求的一个共同的特点是其含有多个处理单元，每个处理单元都既能从事自己的活动，又能协同处理一个大任务。

（2）现实路线（Internet/Intranet下的理解）

随着人们对Internet/Intranet的进一步使用，人们的生活方式和获取信息的方式发生了根本性的变化。在分布式软件的构架和研究方面，那种一味地追求逻辑上完全统一的研究方式已经失败。经验告诉我们，统一协议下的松散式协同工作模式更容易满足人们的需求。目前的分布式研究，不再追求严格的、逻辑上完全统一的系统对分布在各个角落的计算机进行管理，而是用像CORBA或DCOM或CCM等这样的分布式构件模型和中间件技术，结合软件Agent进行基于网络的全方位分布式框架和环境的构架。

由此可见，现实路线是在网络计算机平台上部署开放分布式计算环境，提供开发工具和公共服务，支持分布式应用，实现资源共享和协同工作。在20世纪90年代，工业界普遍追求这一技术路线，产生了一系列行之有效的技术和广为用户接受的产品。当前，人们所说的分布式计算技术通常是指在网络计算机平台上开发、部署、管理和维护以资源共享和协同工作为主要应用目标的分布式应用技术。本书中的讨论也是指这种技术，并称为开放分布式技术。

无论如何，开放分布式技术像分布式技术一样也在不断地发展之中，要给出一个确切的定义是不可能的，但概括起来是指在独立的计算机集合系统中，通过网络通信来开发、部署、管理和维护以资源共享和协同工作为主要应用目标的分布式开发环境。相对于集中式计算模式，开放分布式计算模式在性能价格比、计算能力、可靠性、伸缩性和解决问题固有的分布性上占有明显的优势；而对于桌面计算，分布式计算在数据共享、设备共享、通信、灵活性等方面则显示出了无

可比拟的优势。当然，事物总是一分为二的，开放分布式计算技术也面临着一些急需解决的问题，如没有一个权威统一的标准、缺乏开发分布式系统的支撑软件、网络负载饱和引起的问题、数据安全等问题。

3．开放分布式计算模型

进入90年代，开放分布式计算的实现主要依赖于经典的客户机/服务器计算模型。它将分布式应用分解为客户和服务器两大部分，客户机首先发出请求，服务器在接到客户的请求后提供服务。这种方式不同于主机的计算模型在于：充分利用了客户端计算机的计算能力，每一个客户机都是一个独立的计算单元，有自己的处理器和存储器，负责处理应用系统的显示逻辑，如图形用户界面、信息预处理等，而把复杂的计算，如业务逻辑、数据处理，交给了服务器。通过平衡客户机和服务器的负载以实现分布式资源和信息的共享。目前Internet上最流行的WWW应用就是一种开放分布式的客户机/服务器结构。

WWW应用是一种多层结构，分为客户端、应用服务器、数据库服务器。客户端可以通过浏览器（browser）以标准的通信协议访问分布在网络上的各种多媒体信息，如文本、图像、声音、视频等。浏览器的技术简明易用，一旦用户学会了使用浏览器，也就打开了运用系统上各种信息资源的大门；应用服务器端通过Web服务器提供各种应用服务，包括各种业务处理逻辑；客户机不必关心服务器的具体位置，它可以把分布在网络上的许多服务器当成是一台巨大的“虚拟主机”，各个应用服务可以是并行的也可以是串行的，通过中间件用以消除通信协议、数据库查询语言、应用逻辑与操作系统之间潜在的不兼容问题；数据库服务器就是DBMS（Database Management System），负责管理对数据库数据的读写，迅速执行大量数据的更新和检索。

上述计算模型是在共享分布资源的应用背景下形成的，只是实现完全的分布式计算的一个中间步骤。随着对象模型、构件技术、软件框架技术和Web技术的不断进步，在新的应用需求冲击下，分布式计算开始向分散对等的协同计算方向发展，开放分布式对象技术正成为分布式计算的主流。

对象模型、构件技术、框架技术、软件Agent技术和Web技术的融合彻底改变了系统的构造方法。在开放分布式系统中，对象被用来表示分布的、可移动的、可通信的实体；构件化的软件开发方法使对象被加在网络上、集成在框架和中间件上，达到跨平台的互操作和较高的可伸缩性；Web技术使应用对象可以在Internet这个开放的计算平台上移动。这是一个全新的计算模式，核心是可互操作的对象，即软件对象间可透明地进行相互通信，彼此地位是对等的，可使用对方的服务，而不管这些对象是处于同一编址空间，还是不同的编址空间，或是根本不同的机器上。

在开放分布式对象市场中有3种主要的竞争技术：包含760多个成员的对象管理组织（OMG）的公共对象请求代理体系结构（CORBA）；SUN公司的Java远程方法调用（RMI）和微软公司的分布式构件对象模型（DCOM）。

当前，CORBA和Java以其平台独立性，影响已大大超过了微软的DCOM。事实上Java和CORBA在体系结构上是互补的。CORBA不仅仅只是一种对象请求的中介工具，它还是一个非常完美的分布式对象平台，它使Java应用软件得以在不同的网络中、在不同的操作系统上顺利运行，可以跨越不同语言、不同构件的边界而畅通无阻；Java也不仅仅只是一种新语言，它还是一种可移动对象系统，简化了大型CORBA系统的代码分布处理，它的字节码使对象行为流动化，为CORBA实现流动运行打开了大门。人们发现，Java正是编写客户机和服务器CORBA对象的最佳语言，它内部的多线程机制，无用存储空间收集和出错管理的功能，使其能很容易地编写出健壮的网络化对象代码。作为Internet上影响最大的Web应用，CORBA和Java的加入，不仅大大改善了Web应用的质量，如客户端可直接动态调用服务器上的程序、多对象服务群平衡处理输入的客户请求等，而且对于开放分布式对象的普及起了积极的作用。

开放分布式计算模型广泛应用于目前的软件开发中，从小规模的企业软件的组建，到中大型跨区域、跨国际企业的软件构架，无处不见它的痕迹。基于这种计算模型的技术称为大规模软件构架技术。

在大规模软件构架之中，人们充分利用面向对象的技术，为了达到在分布环境下的软件高度重用，制定了不少标准，其中最有影响的是微软的DCOM、OMG的CORBA和SUN的EJB。然而，在这种异构环境下，不仅包含了不同的硬件，也包含了复杂多样的系统软件和应用软件。经验告诉我们，若要采用集中式管理是绝对行不通的。这种环境的一个最大特点是在松散的基础上讲究一致性和协同性。因而，处于该环境中的用户需要遵循一种协议，进行逻辑上的统一。同时，大量信息的共享加之网络带宽的限制，要求软件具有智能性，同时还应具有移动性，既靠移动的软件处理分布在网络环境下的数据和信息，争取保持数据处于原地不动，从而降低大量数据在网上传递带来的网络堵塞。软件Agent，特别是移动Agent支持下的MAS（Multiple Agent System）为我们提供了一种很好的解决途径。

如果将每一个软件Agent看成一个对象，利用开放分布式构件模型将它们进行整合，将是未来大规模软件开发的主流模式。

本书正是沿着这条思路，从介绍面向对象技术开始，对中间件、分布式构件模型、软件Agent和大规模软件集成等技术做了较详尽的阐述。最后，给出了一个大规模软件构架的应用范型——数字城市。

本书注重软件的构架思想和基本概念，也强调应用技术与方法，它不仅适应

于电子商务、企业信息系统、WebGIS、数字城市乃至数字地球等大规模软件的开发者、应用人员与管理人员阅读参考，也可作为高等院校的高年级学生或研究生的教材或参考书。

由于时间紧，本书难免有许多不足或错误之处，恳请读者批评指正！

本书在编写过程中引用了大量的参考文献，在每章的最后专门列出。在此向被引用文献的所有作者表示最衷心和最诚挚的感谢！

作 者

2003 年 4 月

目　录

第 1 章　面向对象技术

面向对象（OO）是一种认知方法学。它既提供了从一般到特殊的演绎手段（如继承等），又提供了从特殊到一般的归纳形式（如类等）。面向对象基于信息隐蔽和抽象数据类型概念，把系统中所有资源，如数据、模块以及系统看成“对象”，每个对象封装数据和方法，而方法实施对数据的处理。其实，面向对象技术中有关模块化、数据抽象和信息隐蔽等概念也继承了 20 世纪 70 年代软件工程的成果，面向对象技术的贡献是在对象这一更高的层次上进行了抽象。面向对象是一种程序设计方法学。当今计算机正朝着分布式处理、网络化和软件生产工程化的方向发展，而面向对象方法学作为实施这个目标的关键技术之一，显然和计算机领域的总体发展相一致。目前，面向对象技术除了在应用领域的拓广和技术实现的完善等方面正在进行大量的工作外，还试图在一个比“对象”更具有“动态性”和更具有“人性”含义的层次上进行抽象，有人称之为面向智能体（agent oriented）技术。

面向对象的技术把人们自然思考问题的方式作为它的基本原则，为了实现这个原则，在问题域中，必须建立直接表达事物以及事物间相互联系的概念，同时还必须建立适应人们一般思维方式的描述范式。在各种各样面向对象的方法中，对象（object）和消息传递（message passing）是分别表示事物及其相互联系的概念；类（class）和继承（inheritance）是适应人们一般思维方式的描述范式；而方法（method）是允许作用于该类对象上的各种操作。这种对对象、类、消息和方法的程序设计范式的基本点在于对象的封装性（encapsulation）和继承性。通过封装能将对象的定义和对象的实现分开，通过继承能体现类和类之间的关系，以及由此带来的动态绑定（dynamic binding）和实体的多态性（polymorphism），从而构成了面向对象的基本特征。

1.1　对　　象

客观世界的问题都是由客观世界的实体及其间的关系构成，客观世界中问题的集合构成了问题空间或问题域，而实体是问题域的对象。显然，“对象”有大小之分、有“粒度”的概念。在面向对象的程序设计思想中，自然界中的任何事物都被看作对象，世界上的各个事物都是由各种“对象”构成的，这些对象可以属于同一个类，也可以属于不同的类。复杂的对象可由某些简单的对象构成，甚

至这个世界也可以由一些最原始的对象经过层层组合而成。

1.1.1 对象的概念

在面向对象的思想中，对于任何一个对象，都可以采用属性（property）、方法（method）和事件消息（event message）三个方面来描述，这被称为 PME 模型。其中，属性是对象所具有的性质和特征，这些特征可能是看得见摸得着的，也可能是内在的；方法是对象所具有的动作和行为，即使一些无生命的对象也可找出它的方法来，如桌子的倒下、杯子的破裂等；事件是指对象能识别并做出反应的外部刺激，是通过事件消息进行传递的；事件消息是事物关联的桥梁和纽带。

由此可见，对象是一个具有局部属性状态和操作（方法）集合的实体。

本质上，利用计算机解题是借助某种语言的规范，对被计算的实体实施某种动作，以此动作的结果去映射解。我们把计算实体称之为求解域的对象。从动态的观点来看，对象的操作就是对象的行为或方法。问题域中对象的行为是极其丰富的，而解空间对象的行为是极其死板的。因此，只有借助于极其复杂的算法才能操纵解空间对象而得到解。传统的程序设计语言限制了程序员定义解空间对象。而面向对象语言提供了“对象”概念，这样，程序员就可以自己定义解空间的对象。从存储角度看，对象有一片私有存储，其中有数据，也有方法。其他对象的方法不能直接操作该对象的私有数据，只有对象自己的方法才能操纵它。从对象的实现机制看，对象是一个自动机，其中私有数据表示了对象的状态，该状态只能由自己的方法改变它。每当需要改变对象的状态时，只能由其他对象给该对象发送消息，对象响应消息后，按照消息模式找出匹配的方法并执行之。

在面向对象的设计中，对象是应用域中的建模实体，是系统的基本运行单位。换句话说，对象是具有特殊属性和行为方法的实体。对象占有存储空间且具有传统程序设计语言的数据，如数组、字符串和记录等。给对象分配存储单元就确定了给定时刻对象的状态；与每一个对象相关的方法定义了该对象上的操作。所有对象在外观上都表现出相同的特性，即固有的处理能力和通过传递消息的统一的联系方式。

1.1.2 对象的特性

在面向对象的系统中，对象是构成和支撑整个软件系统的最重要、最基础的细胞和基石。每定义一个对象，就增加了一个具有丰富内涵的新的抽象数据类型。对象主要有如下 3 个特性。

（1）模块独立性

从逻辑上看，一个对象是一个独立存在的模块。对外界对象来说，只需要了

解它具有哪些功能，而无需了解它是如何实现的，也无需了解隐蔽在模块内部的信息。也就是说，模块内部状态不会受外界的干扰，也不会涉及其他模块。也就是说，模块之间的依赖性极小，各模块可独立为系统所选用，也可以被程序员重用，而不用担心会影响或破坏其他模块。

（2）易维护性

由于对象的功能实现细节被隐蔽，所以对对象功能的修改、完善等都局限于对象的内部，并不会涉及外部对象，这就使得对象和整个系统变得非常容易维护。

（3）动态连接性

客观世界中各式各样的对象并不是孤立存在的，它们之间存在各种各样的联系。正是这些对象之间的相互作用和相互联系，才使世界变的丰富多彩。在面向对象系统中，通过事件激活机制产生的消息，将各个对象动态地联系在一起，从而形成一个有机的整体。这就是对象的动态连接性。

1.2　类和实例

在面向对象的程序设计中，对象是程序的基本单元。具有相同特性的对象可以归并到一类中去。在面向对象程序设计中，只需定义一个对象类就可以得到若干个实例（instance）对象了。

一个类描述了该类型的所有对象的性质。具体来说，类由方法和属性数据组成，包括外部特性和内部实现两个方面。对象的外部特征是对象类通过描述消息模式及其相应的处理能力来定义的，而对象的内部实现是通过描述内部状态的表现形式及固有处理能力的实现来定义的。类的形式如图 1-1 所示。

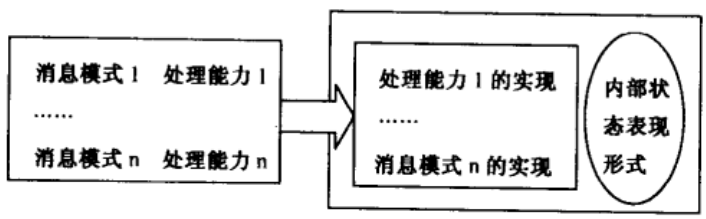


图 1-1　类的形式

从类的形式上来看，类实际上是抽象数据类型 ADTs（Abstract Data Types）的具体实现。数据类型是指数据的集合和作用在其上的操作集合。抽象数据类型的特点是使用和实现的分离，实行封装和信息隐蔽。从使用者的角度看，只要了解该抽象数据类型的规格说明，就可以利用其公共服务来使用这些类型，而不关

心其物理实现，这样，使用者可以在使用中抓住重点，集中精力解决应用问题；从实现者的角度看，把抽象数据类型的物理实现封装起来，有利于编码测试和修改。类是对象集合的抽象。

对象是在执行程序的过程中，由所属的类动态生成，一个类可以生成多个不同的对象。同一个类的所有对象具有相同的性质，即其外部特性和内部实现都是相同的。一个对象的内部状态只能由其自身来改变，任何别的对象都不能改变它。因此，同一个类的对象虽然在内部状态的表现形式上相同，但它们有不同的内部状态，这些对象并不是完全一模一样的。类可形式定义为：

类::= < ID, INH, DD, OI, ITF >

其中，ID：类的标识或类名；

INH：类的继承性；

DD：类的数据结构；

OI：类的操作集合；

ITF：类的对外接口。

继承性可用偏序关系定义为：

INH::= < C, >= >

其中，C：处于继承链上的所有类；

>=：继承关系。

一个类的上层可以有超类（superclass），下层可以有子类（subclass），形成一种类的层次结构，这个层次结构的主要特点是继承性，一个类继承其超类的全部描述。这种继承具有传递性，即 C1 >= C2，C2 >= C3，则 C1 >= C3。所以一个类实际上继承了层次结构中在其上面的所有类的全部描述。因此，属于某个类的对象除了具有该类描述的特性外，还具有层次结构中该类上面所有类描述的全部特性。

抽象类（abstract class）是一种不能建立实例的类。抽象类将有关的类组织在一起，提供一个公共的根，其他一系列的子类从这个根派生出来。抽象类刻画了公共行为的特征，并将这些特征传给它的子类。通常，一个抽象类只描述与这个类有关的操作接口或这些操作的部分实现，完整的实现被留给一个或几个子类。有时抽象类描述了这个类的完整实现，但只有在将这个类和其他类组合在一个新的类中时它才有用。一个抽象类包含了操作接口但没有实现，也就是定义了一种协议，通过从多个父类中继承定义，几个协议可被组合在一起。

综上所述，类是对一组对象的抽象，它将这种对象所具有的共同特征（包括属性和方法）集中起来，由该种对象所共享。

实例是被某一个特定类所描述的一个对象。因此，某一个对象都是某个类的一个实例，而类是对具有相似特征的全部实例的描述。

1.3　消息和方法

对象是问题求解的实体，但是并不是一个孤立的事物。一个系统一定是由相互关联的一组对象组成的，并通过对象之间的相互关联共同完成“整个问题”的求解。

1.3.1　消息

消息就是用来请求对象执行某个处理和回答某些信息的要求，是连接对象的纽带。消息即可以是数据流，又可以是控制流。

在面向对象的系统中，对象间的联系只有通过传递消息进行。对象只有在接收到消息之后才被激活。被激活后的对象代码“知道”如何去操纵它的私有数据去完成该消息所要求的功能。

消息有如下几个性质：

① 同一对象可以接收不同形式的多个消息，产生不同响应；

② 一条消息可以发送给不同的对象，消息的解释完全由接收对象完成，不同的对象对相同形式的消息可以有不同的解释；

③ 与传统的调用返回所不同的是，对于传来的消息，对象可以返回相应的回答信息，也可以不返回，即消息的响应并不是必要的。

1.3.2　消息模式和方法

消息的形式用消息模式（message pattern）来刻画。一个消息模式定义了一类消息。例如，定义“+一个整数”是一个消息模式，那么“+10”、“+20”等都属于该消息模式的消息。

对同一消息模式的不同消息，同一对象所作的解释和处理都是相同的，只是处理结果可能不同。所以，对象应定义一种消息模式和响应的处理方法。消息模式不仅定义了对象接口所能受理的消息，而且还定义了对象的固有处理能力，是定义对象接口的惟一信息，使用对象只需要了解它的消息模式。对象对其消息模式的处理能力即所谓方法，方法是实现消息功能的手段，在C++中方法称为成员函数。

1.3.3　消息的分类

在面向对象的系统中，消息分为公有消息和私有消息两类。对于属于同一对象的一批消息，如果有一部分是由外界对象直接向它发送的，则称之为公有消息；还有一部分则是它自己向本身发送的，则称为私有消息。私有消息不对外公

开，外界不必了解它们。外界对象向该对象发送消息时，只能发送公有消息。

1.4 面向对象的基本特征

面向对象系统的基本特征是继承性、封装性和多态性。

1.4.1 继承性

在现实生活中，对事物分类时并不是一次能分得特别精细，往往是先进行粗分类，然后进一步细分，使类相互联系而形成完整的有机体，这种类之间的关系就是继承。

继承性是自动共享类和子类中方法和属性的一种机制。一个系统类是各自封闭的。如果没有继承性机制，则对象中属性和方法就可能出现大量的重复。

当类 Y 继承了类 X 时，称 Y 是 X 的子类，类 X 是类 Y 的父类。在这种继承关系下，类 Y 由继承部分和增加部分组成，前者是从 X 继承得到的，后者是专门为 Y 编写的。如图 1-2 所示。

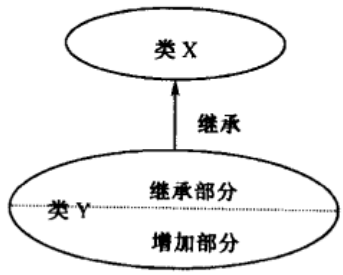


图 1-2 继承关系

在类的继承关系下，可以将类重新定义为：

类 n::=＜类 n 的 ID，

$\bigcup_{i=1}^{n}$ 类 i 本身的数据结构描述，

$\bigcup_{i=1}^{n}$ 类 i 本身的操作实现，

$\bigcup_{i=1}^{n}$ 类 i 本身的对外接口＞

其中，类 i 到类 n 是属于同一类链的，并且类 i + 1 继承类 i 的特性。

概括来说，有继承关系的类之间具有如下几个特性：

① 类间具有共享特征（包括数据和程序代码的共享）；

② 类间具有细微的差别或新增部分（包括非共享的程序代码和数据）；

③ 类间具有层次结构。

继承性有助于开发快速原型，有助于从可重用成分构造软件系统，有助于促进系统的可扩展性。

每一种面向对象的语言都提供一套用于继承的机制。语言级提供的关键词用来表示期望映射的种类，某些映射还需要附加一些信息。例如，如果在进行 X 到 Y 的映射时要重新实现 X 的某一成员，那么必须给出重新实现的代码。

继承关系常称“is a”关系。这是因为当 Y 继承 X 时，由继承性可知 Y 现在已具备 X 的全部性质，所以 Y 即是 X。无疑 Y 也可能包含了 X 中没有的特性，它具有比 X 更多的性质。

引入类的概念，就出现了类的层次结构。一个类的上层可以有超类，下层可以有子类，这样就构成了一种层次结构。在C++ 中超类称为基类，子类称为派生类。

从继承源的角度来看，继承可分为单继承和多继承。所谓单继承就是每个类只继承一个基类的特性；多继承则是指派生类中继承了不止一个基类的特性。

1.4.2　封装性

封装是一种信息隐蔽技术，用户只能看到对象的封装界面，对象内部对用户是隐蔽的。封装的目的在于将对象的使用者和对象的设计者分开，使用者不必知道行为实现的细节，只需用设计者提供的消息来访问该对象，从而更有利于使用者将精力集中在解决问题上。

封装的定义为：

① 一个清楚的边界，所有对象的内部软件范围被限定在这个边界内；

② 一个接口，这个接口描述这个对象和其他对象之间的相互作用；

③ 受保护的内部实现，这个实现给出了对象提供的功能细节，这些细节不能在定义这个对象的类的外面访问。

封装的概念和类的说明有关，但它同样提供如何将一个问题的各个实体组装在一起的求解过程。封装的基本单位是实体对象，这个对象的性质用它的类说明来描述，这个性质被具有同样类的其他对象共享。借助封装这个定义，一个类的每一个实例在问题求解中是一个单独的实体。

由此可见，封装应具有如下几个条件：

① 具有一个清晰的边界，对象所有的私有数据、内部程序细节都被限制在这个边界内；

② 具有一个接口，这个接口描述该对象和其他对象之间的相互作用、请求和响应，即消息；

③ 对象内部的实现细节受封装壳保护，其他对象不能直接修改该对象所拥

有的数据和消息。

封装本身体现了模块化，把定义模块和实现模块分开，使软件的可维护性、可修改性大为改善。

面向对象的语言以对象协议（protocol）或规格说明（specification）作为对象的外界面，它告诉一个对象可以对外界做什么。协议说明该对象所能接收的消息，在对象的内部，每个消息响应一个方法，方法实施对数据的运算。对方法的描述是协议的实现部分，叫类体。协议实际上是一个对象所能接收的所有共有消息的集合。

在面向对象的概念中，继承和封装是两个矛盾体。有了封装机制，对象之间只能通过消息传递进行通信，而继承机制的引入似乎削弱了封装性。

实质上，对象的封装性和继承性并没有实质性的冲突。在面向对象系统中，封装性主要是指对象的封装性，即将属于某一类的一个对象封装起来。

从另一个角度来看，继承性和封装性有一定的相似性，即它们都是一种共享代码的手段。继承是一种静态共享代码的手段，通过派生类完成对象的创建，可以接收某一消息启动其基类所定义的代码段，从而使基类和派生类共享了这一段代码。而封装机制所提供的是一种动态共享代码的手段，通过封装，可以将一段代码定义在一个类中，在另一个类所定义的操作中，可以通过创建该类的实例，并向它发送消息而启动这一段代码，同样也达到了共享的目的。由此可见，在引入了继承机制的面向对象中，对象依然是封装得很好的实体，其他对象与它通信的途径只有一条，那就是发送信息。类机制是一种静态机制，不管是基类还是派生类，对于一个对象来说，它仍然是一个类的实例，它丝毫没有影响对象的封装性。

1.4.3 多态性

多态性是面向对象的又一重要特性。多态性是指允许不同类的对象对同一消息作出响应。也就是说，不同的对象接收到相同的消息时产生不同的动作。

在大多数面向对象的语言中，如果类 P 是子类 S 的父类，则子类 S 的对象 s 可以用在父类 P 的对象 p 被使用的地方，这就意味着公共的消息集（即操作）可以送到类 P 和类 S 的对象上。当同样的消息可以送到一个父类的对象和它的子类的对象上时，就产生了多态性。

多态性具有静态类型和动态类型。动态类型可以在程序执行期间在实例之间进行变化，即在程序执行期间内可以改变类型。静态类型是在程序的上下文中由实例说明决定。

由此可见，对象的多态性是指在一般类中定义的属性或服务被特殊类继承之后，可以具有不同的数据类型或表现出不同的行为。这使得同一属性或服务名在

一般类型及其各个特殊类中具有不同的语义。

在C++中，函数或运算符的重载就是对多态性的具体实现。

有些评论者没有把多态性列入 OO 方法的基本特征或列入 OOPL（面向对象的编程语言）的必备功能，而是看作一种比较高级的功能。多态性的实现，需要 OOPL 相应的支持。与多态性的实现有关的语言功能有：

① 重载（overload）：在特殊类中对继承类的属性或服务进行重新定义；

② 动态绑定：在运行时根据对象所接收的消息动态地确定要连接到哪一段程序代码；

③ 类属（generic）：服务参数的类型可以是参数化的。

1.5　面向对象的软件生存周期

用面向对象方法开发软件，软件生存周期中各个时期有其特有的要求和规则。一般来说，面向对象的软件开发过程与典型的瀑布模型和典型的原型模型有所不同，它是一种迭代、渐增式的开发过程，每个阶段都可以相互反馈，如图 1-3 所示。

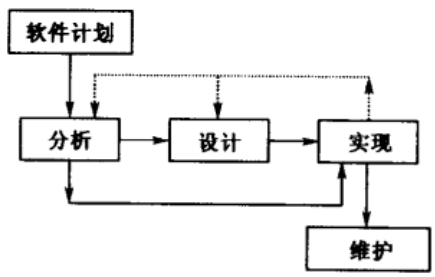


图 1-3　面向对象的软件生存周期

（1）面向对象的分析

在面向对象的方法中，软件分析的基本任务是从实现问题空间抽象出对象空间。具体地说，面向对象分析的核心是从问题空间选出词汇建立类和对象的模型。在分析阶段主要问题集中在如何找出关键抽象，明确问题中存在哪些数据实体，它们的意义是什么，暂时并不考虑它们是如何实现的。分析阶段一般包括定义问题边界、问题空间的关键抽象、类的抽象等三个方面。

（2）面向对象的设计

面向对象方法设计阶段的主要任务是对问题空间进行关键抽象和再分解，从而给出类的层次结构以及对象之间的关系。将设计的结果反馈到分析阶段进行修正，直到关键抽象足够简单而不需要再分解为止。该时期的任务还包括对象/类

的关系的具体化，确定方法和函数的实现算法等。

（3）面向对象的实现

软件实现是指在设计的基础上进行的编码、测试、集成组装的迭代过程。对一个复杂的系统，不可能将问题空间的每一个细节都分析和设计得完备无缺才开始编码，何时进行编码工作实现，一般来说，主要由问题空间中最基本的设计是否已完成来决定。实现阶段的结果也要返回到分析和设计阶段。具体任务可归纳如下：

① 识别问题及其解中出现的对象；

② 根据对象的共同点和不同点对它们进行分类；

③ 设计出对象间相互作用的消息；

④ 实现执行对象间相互作用的算法的方法。

参考文献

邵维忠，杨芙清．1998．面向对象的系统分析．北京：清华大学出版社

汤庸．1998．结构化与面向对象软件方法．北京：科学出版社

汪成为，郑小军，彭木昌．1992．面向对象分析、设计与应用．北京：国防工业出版社

第 2 章　开放分布式处理及软件体系结构

分布式计算技术是近 20 年来影响计算技术发展的最为活跃的因素之一，它的发展经历了两种不同的技术路线：理想路线和现实路线。

理想路线试图采用全新的分布式操作系统，全面管理基于网络（包括局域网、城域网和广域网）的计算机，呈现给用户单一的系统视图，这些技术总称为分布式计算机系统技术。在 20 世纪 80 年代，学术界普遍追求这一目标，尽管产生了许多技术成果和实验系统，但却没有被用户和市场接受。对于分布式计算机系统的定义一直不太明确的一个主要原因是，不知采用什么样的模型来构造、管理和协调使用基于网络中的各个计算机系统。但是，对于分布式计算机系统，当时追求的一个共同的特点是其含有多个处理单元，每个处理单元都既能从事自己的活动，又能协同处理一个大任务。

现实路线是在计算机网络平台上部署分布式计算环境，提供开发工具和公共服务，支持分布式应用，实现资源共享和协同工作。在 20 世纪 90 年代，工业界普遍追求这一技术路线，产生了一系列行之有效的方法和广为用户接受的产品。当前，人们所说的分布式计算技术通常是指在网络计算机平台上开发、部署、管理和维护以资源共享和协同工作为主要应用目标的分布式应用技术。本书中的讨论也是指这种技术，并称为开放分布式处理技术。

目前，分布式技术的发展已进入到了第三代时期，基于大规模软件体系结构的智能化协同工作、自主的多 Agent 体系结构模型和面向 Agent 的拟人化分布式交互环境是它的主要特点。

起初，人们把软件设计的重点放在数据结构和算法的选择上。当前，随着软件系统规模越来越大、分布异构性越来越复杂，整个系统的结构和规格说明显得越来越重要。现有的软件工程方法对此显得力不从心。特别对于大规模的复杂软件系统来说，对总体的系统结构设计和规格说明比起对计算的算法和数据结构的选择已经变得明显重要得多。在这种背景下，人们认识到软件体系结构的重要性，并认为对软件体系结构的深入研究将会成为提高软件生产率和解决软件维护问题的新的最有希望的途径。

其实，自从软件系统首次被分成许多模块，模块之间有相互作用，组合起来有整体的属性，就具有了体系结构的概念。好的开发者常常会使用一些体系结构模式作为软件系统结构设计策略，但他们并没有规范地、明确地表达出来，这样就无法将他们的知识与别人交流。软件体系结构是设计抽象的进一步发展，满足

了更好地理解软件系统，更方便地开发更大、更复杂的软件系统的需要。

事实上，软件总是有体系结构的，不存在没有体系结构的软件。体系结构（architecture）一词在英文里就是“建筑”的意思。把软件比作一座楼房，从整体上讲，是因为它有基础、主体和装饰，即操作系统之上的基础设施软件、实现计算逻辑的主体应用程序、方便使用的用户界面程序。从细节上来看每一个程序也是有结构的。早期的结构化程序就是以语句组成模块，模块的聚集和嵌套形成层层调用的程序结构，也就是体系结构。结构化程序的程序（表达）结构和（计算的）逻辑结构的一致性及自顶向下开发方法自然而然地形成了体系结构。由于结构化程序设计时代程序规模不大，通过强调结构化程序设计方法学，自顶向下、逐步求精，并注意模块的耦合性就可以得到相对良好的结构，所以，并未特别研究软件体系结构。

虽然，软件体系结构来自软件工程，但最近几年对它的研究已完全独立于软件工程，成为计算机科学的一个最新的研究方向和独立的学科分支。解决好软件的重用、质量和维护问题，是研究软件体系结构的根本目的。

2.1 开放分布式处理技术 RM-ODP

早期的计算机通信是同一类计算机间的通信，随着网络的发展及分布式系统的日益流行，大量异构网络及各计算机厂商推出的软硬件产品造成在分布式系统的各层次上存在着广泛的互操作问题。而分布式操作和应用接口的异构性严重影响了系统间的互操作性。要实现在异构环境下的信息交互，就需要重新确立一种不基于特种操作系统、特种机型或厂家的全新网络体系结构，即所谓开放分布式处理系统。

开放分布式处理技术把异构（包括软件异构和硬件异构）分布信息系统组合在一起，使得跨组织的信息协作成为可能。要完成这一任务，开放分布式处理系统必须遵循一个公共的体系结构。而制定开放标准是建立开放分布式处理系统的最直接途径，也是异构信息系统间实现可移植性和互操作性的基本策略。20 世纪 80 年代前期，国际上已有的提供计算机系统之间互连能力的应用程序互操作框架模型主要是 ISO 的 OSI-RM（OSI 七层模型）和 TCP/IP 等。90 年代初，经过国际标准化组织 ISO/IEC 和 ITU-T 的共同努力，开发出了开放分布式处理参考模型 RM-ODP。

然而，ISO 的开放系统互连模型，以及 ITU-T 以前制定的各种标准主要解决的是对等通信问题，它们使用标准通信协议（相当于一个通信协议栈）来描述系统通信接口的行为，无法表达和控制系统内对象的任何配置。其主要存在以下难以解决的问题：

① 应用所需的服务由哪些对象提供；

② 应用所需的服务位于何处；

③ 如何获得所需的服务；

④ 如何组织和配置各种网络服务，使分布式应用能协同工作；

⑤ 分布构件的执行和交互需要什么样的分布式支撑环境。

为了解决上述问题，需要有一种能兼容异质成分，并能够表达分布式应用构件之间互操作的通用模型，国际上将这样的分布式处理称为开放式分布处理 ODP（Open Distributed Processing）。ODP 的目标是分布性、可移植性、互操作性和透明性。

2.1.1　RM-ODP 框架与理念

RM-ODP 是一种“元标准”，不仅自身可以作为一种标准，还能够协调和指导不同应用领域的 ODP 标准的开发。即它是一种框架，在其上可以实现不同的“分布式计算环境”。开放分布式处理系统相对于传统的集中式系统而言有许多新的特色，如异构性、自治性、演进性和可移植性等。ISO 和 ITU-T 共同提出的 RM-ODP 模型是一项试图解决开放分布式环境下软件接口问题的新兴技术。它是一个通用的框架结构，用来指导对分布式环境下的开放系统进行规范描述和开发。同时，这两个组织也意识到，不可能通过一种单一的公共基础平台来提供分布式系统所需的各种特性，而需要选择不同的通用平台构件和构架满足各种特定需求。因此，ODP 框架也应能对构件以及反映这些构件之间关系的构架进行描述。ODP 的主要任务就是定义满足上述需要的通用框架，即能表达“应用互操作”和“分布式平台服务”的框架。

相比较而言，OSI 七层模型只是 RM-ODP 系统中的通信技术。

RM-ODP 的能力范围主要包括：

① 支持多厂商；

② 可使用异构技术实现系统；

③ 分布式系统在范围上可扩展到全球；

④ 分布式系统可平滑更新；

⑤ 不同应用之间的相互操作和数据共享；

⑥ 减少开发和操作的开销。

RM-ODP 的目标是：

(1) 互操作性

主要指提供不同系统之间的信息在语法和语义上的交换，以及系统间功能服务的方便使用，特别是资源的动态发现与挖掘。

(2) 可移植性

提供应用程序在分布环境系统中的迁移能力，并且不破坏应用所提供的或正在使用的服务。这种迁移包括静态的系统重构与重新安装以及动态的系统重构。

(3) 分布透明性

分布透明性的主要作用是屏蔽了由于分布所带来的系统复杂性。在编程方面更是如此，分布透明性使应用编程者不必关心系统是分布的还是集中的，从而可以专心设计具体的应用；另外，应用编程者可以选择自己所需要的透明性，而其他透明性则由应用本身来实现。RM-ODP 提供了以下 8 种透明性：

① 访问透明性（access transparency）。它能屏蔽数据表示和调用机制的差别，使用同样的方式存取数据，使得不同对象之间可以协作。该透明性解决了异构系统协作时所遇到的大部分问题。

② 位置透明性（location transparency）。它能屏蔽分布式环境中对象的空间位置信息。该透明性使编程者不必知道对象的实际物理地址，而通过它的逻辑名来引用对象。

③ 迁移透明性（migration transparency）。即系统为使资源均衡而把一个服务对象迁移到另外的位置，而无需用户干预，这种迁移对用户透明。

④ 失败透明性（failure transparency）。它能将对象的失败和可能的恢复事件隐藏在对象内部，使系统具有容错性能。如果提供了这种透明性，设计者可以在一个理想而健壮的环境中工作而永远不会发生错误。

⑤ 重定位透明性（relocation transparency）。即使在交互中的对象已经被搬家或替换，并不影响系统的运行，即改变一个接口的位置不会影响与之联编（binding）的其他接口。

⑥ 复制透明性（replication transparency）。该透明性能屏蔽对象的复制，可以提高系统的性能。例如，系统中可能同时有多个行为相同的对象支持某一接口，而用户不知道有多少个对象存在。

⑦ 持久透明性（persistence transparency）。它能够屏蔽系统中对象资源的变化，对象的激活、撤消和再激活等。当系统不能为某一对象继续提供处理能力、存储能力和通行能力时，撤消和再激活对象，用于维持对象的持久性。

⑧ 事务处理透明性（transaction transparency）。在分布式环境中，事务处理透明性能屏蔽一组对象间开展活动要进行的协调，使一组对象发生作用的次序不影响结果的一致性，确保信息的完整性。

RM-ODP 结构是基于面向对象范型（OO paradigm）的，主要体现在：

① 对象是分布和管理的基本单位；

② 对象可拥有多个接口，通过接口向外界提供服务；

③ 接口具有类型，强调基于类型检查的后联编机制（*late-binding*）。

可以说，RM-ODP 结构是一种面向对象分布系统的全景图。面向对象技术

简化了分布式系统的设计，具有良好的复杂处理能力，其 3 个关键特性（封装性、多态性和继承性）较好地适应了分布式系统。RM-ODP 模型在分布式系统的设计及开发中鼓励采用面向对象的模型和方法学，其交互的接口是面向服务（service-oriented）的。

RM-ODP 研究的重点内容有：

① 构件；
② 系统构成成分及其接口；
③ 接口的标准化；
④ 基于接口规范的交易（trading）和联编服务；
⑤ 互操作，应用构件之间逻辑关系的任意配置；
⑥ 集成，通过分布构件的偶合，来提供特定的服务；
⑦ 可移植性，即对分布式平台的标准化；
⑧ 透明性，即对应用程序屏蔽分布环境的细节；
⑨ 多媒体，即针对媒体的应用采用一致的建模框架。

2.1.2　RM-ODP 的视点模型

对于开放分布式系统来说，一方面是，由于任何分布式应用系统都有许多不同类型的用户，所有对此系统感兴趣的用户，由于他们所担任角色和观察角度的不同，他们所关心的问题的侧重点和对系统的观点也有所不同；另一方面是，开放分布式系统由于相当大而难以分析、难以划分和设计、难以管理。为了解决 ODP 系统的复杂性，RM-ODP 引入视点的概念来对不同的问题分而治之。视点是系统在不同侧面的投影和抽象，使用者通过抽象而抓住了事物的主要矛盾，抓住了处理问题的本质。所有视点的综合构成了对整个系统的全面描述。视点不仅是理论抽象，更主要被作为一种工具，被应用在不同粒度的系统分析和设计过程中。

RM-ODP 的视点包括 5 个模型，每个视点模型代表了对原始系统的不同角度和不同侧面的抽象，并采用相应的“视点规范语言”，以便对视点进行标准化研究。视点语言定义了一个结构化的概念集合和附加在其上的规则，用于从一个特定的视点描述 ODP 系统。亦即每个视点都有相应的视点规范语言，视点规范语言是一组原理定义和构造规则的集合，原理规定了本视点中描述分布式系统的词汇，规定了应用这些词汇构造系统规范的语法。系统的视点规范是指在满足其构造规则的情况下应用该视点原理。如果一种现存的语言在语法和语义上能够表达该视点的原理和规则，则都可以用于描述该视点的系统规范。下面对各类视点分别进行描述。

1. 企业视点（enterprise viewpoint）

企业视点用于描述分布式系统的总体目标、范围和策略，它关心的是企业的业务活动。即描述系统要为企业完成什么样的功能，解决企业管理者和决策者所遇到的各类问题。企业视点描述的具体内容包括：

① 企业中的哪种角色（roles）使用该系统，其行为（activities）是什么；
② 角色之间的约束和限制关系（contract）；
③ 有哪些主动对象（agent）；
④ 被动对象（artifacts）被企业中的哪种角色利用；
⑤ ODP 系统同其他环境如何相互作用；
⑥ 企业的组织结构如何；
⑦ 与角色使用有关的企业安全和管理政策等。

这里的企业不仅是用户，也可以是拥有边界权限的信息系统资源。企业可覆盖组织的一小部分、一个组织、直至多个组织。

2. 信息视点（information viewpoint）

信息视点提供对分布式系统中的信息、对信息施加的操作以及信息间关系模型和信息流的描述。即用于向信息管理者和信息工程师等描述企业对 ODP 系统信息的需求。

信息视点采用信息建模技术，集中针对企业中的信息内容，定义开放分布式系统中信息和信息处理的语义，明确描述系统中的信息结构、施加于这些信息结构上的约束和处理以及信息在系统中的流动等内容。信息视点描述的主要内容有：

① 定义企业中信息的元素结构；
② 信息元素的特征属性；
③ 对信息元素之间的关系规则进行描述；
④ 描述信息的变化和导出以及相应的规则；
⑤ 信息流建模；
⑥ 描述企业中用户可见信息和信息处理形式；
⑦ 进行 ODP 系统中各个自治部分的逻辑划分。

3. 计算视点（computation viewpoint）

计算视点是从系统设计者和编辑者的角度出发，将系统的功能分解一系列独立执行功能的对象（计算对象），对象之间通过预定义的接口进行交互。每个计算对象都实现一个或多个接口，接口的作用是向其他对象提供其可见的操作，在

接口上交互的对象都各自扮演着不同的角色。也就是说，计算视点定义了 ODP 系统的对象，描述了对象中发生的动作和对象间的交互过程。

计算视点模型的核心内容是服务。提供服务和接收服务的双方分别为服务器和客户机，而对象（或构件）则是服务器和客户机的组成部分。一个对象可以提供或使用一个以上服务。对象经过封装后，可以被安装、升级、替换或重定位，但不影响其他对象。

4．工程视点（engineer viewpoint）

工程视点重点解决通信设计者所遇到的问题，描述实现分布式对象之间的交互机制。工程视点规范描述了一个网络基础设施的定义、系统的结构和所需的分布式透明服务，为操作系统和通信专家提供一个 ODP 系统的抽象描述。工程视点描述的主要内容有：

① 描述运行 ODP 系统功能的抽象体系结构；

② 抽象地描述局部的和物理上分布的系统资源；

③ 定义支持 ODP 功能的各种对象的任务；

④ 确定不同对象之间的参考点。

企业视点模型、信息视点模型和计算视点模型都是用来描述分布式应用语义的。相比较之下，如果语义本身没有分布式的需求，工程视点模型不再重点考虑应用语义。工程视点对组成分布式处理平台的服务和机制进行抽象，侧重于描述分布式处理平台能对应用提供哪些服务以及实现这些服务的机制。

5．技术视点（techology viewpoint）

技术视点重点解决系统实现者所遇到的问题，它关心的是系统的组成单元细节，着眼于具体实现技术对象的选择，如操作系统、计算机网络、硬件部件、网络开发平台、数据库系统、程序开发语言等。技术视点模型是其他视点描述和系统实现之间的桥梁和纽带，即描述了如何根据其他视点的描述规范，选择和配置合适的技术对象来实现 ODP 系统。

2.1.3　RM-ODP 的功能

ODP 在工程视点中描述了分布式支撑平台的基本组成要素和为应用提供的一系列分布透明性。同时也定义了一组通用功能来支持该平台和实现分布透明性。ODP 的通用功能包括如下 4 个组成部分。

（1）管理功能

实施对工程模型中分布式平台基本组成要素的管理，包括对进程和线程的管理、时间的访问和管理、通道的建立以及工程对象接口引用的创建、容器模板的

实例化和容器的删除等。

管理功能又包括节点管理、对象管理、族管理和容器管理功能。

(2) 协调功能

通过消息传递等方式协调多个对象之间的交互，以完成某种特定的功能。例如，时间通知功能实现分布式环境下的异步事件分发机制，事件的生产者向该功能提供事件信息，事件的消费者则向该功能登记希望接收哪类事件。通过事件通知功能，将实时到来的事件发送到对其感兴趣的所有事件消费者。

协调功能又包括事件通知、断点检查和恢复、激活和取活、复制、迁移、事务和工程接口引用跟踪等功能。

(3) 仓库功能

提供了分布式系统所需要的动态和静态信息的存储和检索功能。它包括存储、信息组织、重置、类型仓库和交易功能。交易功能是 ODP 最为核心的功能。当一个服务器想让所有潜在的客户知道它所能提供的服务时，它向交易器登记一个服务，此时该服务器就成为了一个潜在的可用的服务器；当客户请求某一个服务时，它向交易器登记自己的请求，交易器根据客户输入的请求，查找出能够满足这一要求的服务器。一旦成功，则向客户返回有关服务器的信息，以供在客户和服务器之间建立连接并进行交互。

(4) 安全功能

提供对分布式系统进行安全管理的机制。其包括访问控制、安全审计、完整性、机密性、不可否认和密钥管理等功能。

国际上许多研究机构或标准组织开展了大量的 ODP 方面的研究，并推出了一些支持 ODP 的开发环境，如 OSF/DCE、Microsoft/DCOM 和 OMG/CORBA 等。但与 RM-ODP 的整个框架规范要求相比，还有许多问题需要进一步的研究和解决。

2.2 分布式对象软件体系结构

软件体系结构属于软件工程方面较新的内容。软件体系结构研究的主要内容涉及软件体系结构描述、软件体系结构风格、软件体系结构评价和软件体系结构的形式化方法等。

近几年来，面对日益复杂的软件系统，人们开始认识到，要真正实现软件的工业化生产方式，达到软件产业发展所需要的软件生产率和质量，软件复用是一条现实可行的途径。在软件设计过程中人们所面临的问题不再是考虑软件系统的功能问题，而是面临要解决更难处理的非功能性需求，如系统性能问题、可适应性问题、可靠性问题等。这就要求在系统设计的最初阶段决策系统设计的总原则

和确定整个系统的总体框架，以指导系统设计的开展，保证开发工作能达到项目的当前目标，并能在软件系统的整个生命期中保持系统体系结构可以很方便地进行维护和调整以适应所发生的变化。软件体系结构在大规模软件构架技术中，对于软件宏观结构的控制和软件重用（包括软件框架的重用）显得更为重要。

软件重用是指在两次或多次不同的软件开发过程中重复使用相同或相近软件元素的过程。软件元素包括程序代码、软件的体系结构、测试用例、设计文档、设计过程、需求分析文档甚至领域知识。通常，把这种可重用的元素称作软构件，可重用的软件元素越大，我们就说重用的粒度越大。

使用软件重用技术可以减少软件开发活动中大量的重复性工作，这样就能提高软件生产率，降低开发成本，缩短开发周期。同时，由于软构件大都经过严格的质量认证，并在实际运行环境中得到校验，因此，重用软构件有助于改善软件质量。此外，大量使用软构件，软件的灵活性和标准化程度也可望得到提高。

代码重用是面向对象技术的主要优点之一，通过重用类库中已有类的对象可以提高程序员的软件生产能力和生产效率。然而，对象作为构成应用程序的不可缺少的部分，完全不能把握程序的体系结构。由此可见，面向对象技术、类和类库不具备提供软件高度重用的能力。实际情况是，大量的应用程序，特别是同一领域中的应用程序结构常常非常相似，不同的程序员使用不同的技术去把握和实现这些相似的结构。可见，软件体系结构的重用亦是提高软件生产效率的关键。然而，网络的快速发展对分布式软件的开发提出了更高的要求，人们迫切需要使一般的编程人员也能驾驭的软件体系结构，进而快速有效地构造开放分布式系统；加之软件体系结构并没有通过通常的面向对象技术而得到重用。为此，引入了构件（component）的思想。

2.2.1　软件构件

一般认为，构件是指语义完整、语法正确和有可重用价值的单位软件。结构上，它是语义描述、通信接口和实现代码的复合体。简单地说，构件是具有一定的功能，能够独立工作或能同其他构件装配起来协调工作的程序体，构件的使用同它的开发、生产无关。从抽象程度来看，面向对象技术已达到了类级重用（代码重用），它以类为封装的单位。这样的重用粒度还太小，不足以解决异构互操作和效率更高的重用。构件将抽象的程度提到一个更高的层次，它是对一组类的组合进行封装，并代表完成一个或多个功能的特定服务，也为用户提供了多个接口。整个构件隐藏了具体的实现，只用接口对外提供服务。

（1）构件的分类

如果把软件系统看成是构件的集合，那么从构件的外部形态来看，构成一个系统的构件可分为 5 类：

① 独立而成熟的构件。独立而成熟的构件得到了实际运行环境的多次检验，该类构件隐藏了所有接口，用户只需用规定好的命令进行使用。例如，数据库管理系统和操作系统等。

② 有限制的构件。有限制的构件提供了接口，指出了使用的条件和前提，这种构件在装配时，会产生资源冲突、覆盖等影响，在使用时需要加以测试。例如，各种面向对象程序设计语言中的基础类库等。

③ 适应性构件。适应性构件进行了包装或使用了接口技术，把不兼容性、资源冲突等进行了处理，可以直接使用。这种构件可以不加修改地使用在各种环境中。例如 ActiveX 等。

④ 装配的构件。装配的构件在安装时，已经装配在操作系统、数据库管理系统或信息系统不同层次上，使用胶水代码（glue code）就可以进行连接使用。目前一些软件商提供的大多数软件产品都属于这一类。

⑤ 可修改的构件。可修改的构件可以进行版本替换。如果对原构件修改错误、增加新功能，可以利用重新“包装”或写接口来实现构件的替换。这种构件在应用系统开发中使用得比较多。

构件是用于复用的软件实体。根据复用阶段和复用方式的不同，与之相对应的构件表现形式也不同。目前的构件以二进制目标码构件为主，这种构件复用的特点是无需修改代码，复用方式最为直接。实际上，这也正是以往应用程序对操作系统和软件支撑平台的复用方式。

(2) 构件定义的内涵

构件是组成软件的基本单位，构件的定义包含 3 个方面：

① 构件是可复用的、自包含的、独立于具体应用的软件对象模块；

② 对构件的访问只有通过其接口进行；

③ 构件不直接与别的构件通信。

(3) 构件的性质

构件的性质如下：

① 可插用性。是现成的可打包的软件部件，可直接发行或购买。

② 独立性。即不能与别的构件直接通信，这是构件可维护和可升级的必要条件。此外，独立性使得构件易被用于分布环境中。

③ 粒度性。一个构件是整个应用程序中的一个部件，是为执行某种特定的任务而设计的，构件的粒度一般与其功能的大小相一致。即粒度小，易维护；粒度大，功能强。

④ 自我描述性。构件通常用与实现无关的接口定义语言，向系统的其余部分描述和说明它所提供的服务。

⑤ 框架的目标性。构件通常是为一种特殊的框架所建立的，这种构件不能

在其他框架中运行。如 Java Bean 构件不能直接用于 ActiveX 环境。

⑥ 对象总线的目标性。构件直接与某一对象总线相关联，一般不能与其他的对象总线一起使用，不同对象总线之间的互操作只能通过网关来完成。

(4) 构件的复用者和被复用者之间的关系

从运行过程看，构件的复用者和被复用者之间存在 3 种分布关系：

① 处于同一运行空间中；

② 处于同一节点的不同的运行空间中；

③ 处于不同的节点上。

为了追求最大程度的复用，一个完善的构件模型必须对这 3 种关系进行一致化处理，使自身具有涵盖不同分布关系下构件复用的能力。前两者复用比较容易实现，而对于处于不同节点上的构件复用较为困难，因而是构件研究的重要内容。与一般构件相比较，分布构件的更突出之处在于构件间的连接。即只有当客户方构件主动向服务方构件发出连接请求时，才建立起连接关系。分布构件间的连接除了采用适配器进行静态建立之外，还可以在运行过程中动态地建立。

(5) 构件描述的内容

为了让复用者能使用一个分布式的构件，构件描述的内容包括：

① 属性。描述了构件的特征。属性值对外可以读出，也可以修改。

② 功能接口。是对构件功能与活动的一种定义，也是构件的方法，即构件向外提供的服务。

③ 依赖关系。指出构件在实例化时所依赖的其他构件的特定接口，它是构件完成其他部分所必需的内容。

构件可分为源代码构件和二进制代码构件。构件复用有白盒和黑盒两种形式。黑盒是指不作修改的直接引用；白盒指进行适应性修改的引用。源代码在大多数情况下是指适应性修改地引用；二进制代码构件的复用只能是黑盒复用方式，通常只能了解构件的接口和属性等信息。

源代码构件的复用通常在具体语言一级实现，而二进制只能在运行级上实现，不需要重新编译即可立即运行，因此容易实现即插即用的复用和分布环境中的复用。目前已实现的主要规范 DCOM、CORBA 和 JavaBean 都是二进制级上的代码复用。

(6) 基于构件的开发

基于构件的软件开发通常包括构件获取、构件分类和检索、构件评估、构件的适应性修改以及将现有构件在新的语境下组装成新的系统。

构件获取可以有多种不同的途径：

① 从现有构件中获得符合要求的构件，直接使用或作适应性修改，得到可重用的构件；

② 通过遗产工程，将具有潜在重用价值的构件提取出来，得到可重用的构件；

③ 从市场上购买现成的商业构件，即 COTS 构件；

④ 开发新的符合要求的构件。

2.2.2 软件框架

在对象技术十分成熟的当今，对象技术在实现信息隐藏和数据抽象方面是相当成功的，但在代码的高度重用方面特别是分布环境下的代码重用方面显得有点笨拙和不尽人意。

确实，类库实现了细颗粒的对象，它们能被各种程序分享。但是这种细粒度的对象只占全部应用程序的一部分。它们完全不能把握将这些对象拼装在一起所需的逻辑联系，即完全不能把握软件的体系结构或框架。

基于过程的应用程序包含大量相互调用的过程，这些过程通常是粗颗粒的对象，因此不能在别的应用程序中重用。更进一步将这些过程拼装在一起，形成一个应用程序所需的逻辑关系被分散到应用程序的各个部分，不能由任一函数或过程所把握。

面向对象的应用程序用细颗粒的对象取代了粗粒度的过程，用方法调用取代了过程调用。这些对象由于呈现小细颗粒特征，可以在别的应用程序中得到广泛重用。但是对象仍然是被动的，它仍然需要一种结构将它们连接起来以表达它们之间的逻辑关系。由于这种结构不能由任一对象把握，故它也不能为其他相似的应用程序所分享。

随着应用领域的发展和完善，某些带有整体应用性的结构被逐步“固定”下来，形成了特定的系统结构，它包括系统的基本构成单元和关系，这就是框架的原始形成。框架更多的是关于应用领域问题已建立的体系结构，所以也称作应用框架。从设计模型的角度看，应用框架是一个给定应用领域中的完全的软件系统模式，是大粒度的可复用构件。框架的功能在于：把握许多相似应用程序的结构，为运行一批对象提供一个有组织的环境。构件不再相互调用方法，而是通过框架调用方法。不仅构件可以与别的应用程序分享，而且框架本身也可以被其他应用程序所分享。如图 2-1 所示。

从组成上看，框架是一个等待实例化的完整的系统，它定义了一个软件系统的族和关系，并提供了创建它们的基本构建模块。框架也定义了设置具体功能更改的插件位置。本质上，框架对相似问题集提供了一种统一的解决方案。这一层次上的代码重用远超出了基于被动的类库的代码重用。框架的最终目的是能动态地组装构件，实现软件的即插即用。

框架表示应用程序的一部分，这部分由领域内的专家设计并编码和调试，开

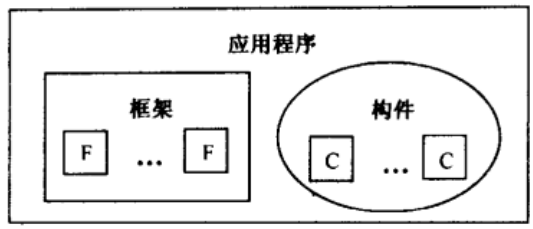


图 2-1　基于框架的应用程序

发者只需在框架上再添加上附加值的代码即可。这些附加代码占整个应用的 20%，而框架占 80%。框架很像类，只能设计和测试一次，开发者通过向框架中添加区别于其他相似程序所需的动作来建立特定的应用程序。

从“模板”的角度看，当使用一个应用框架创建一个具体的软件系统时，要根据用户的特别需求，对框架的热点进行专门化。这通常使用模板-挂钩模型（template-hook model）和元模型（meta-pattern）。模板方法是定义类的抽象行为、控制流或者类之间关系的一种方法，通常用于实现应用程序的冻节点。策略和挂钩方法通常用于实现热点，可以是抽象方法、常规方法或模板方法。

2.2.3　对象总线

对象总线的概念则把构件和框架的能力扩展到了开发基于网络的应用。它使数百万独立的软件单元在异构环境下达到无缝的交互操作。对象总线最著名的是 OMG 的 CORBA 和微软的 DCOM。

两个相互通信的对象并不需要相互理解对方的语言。对它们来说，只要能够理解一种共同的语言即可。在软件构件结构中，这种公共语言是一种对象总线语言。对象总线为所有对象提供了一种向总线描述自己的机制，这种机制通过一种说明性的、与实现无关的接口语言（IDL）提供。任何应用程序、软件系统和工具只要有符合 IDL 标准规范的接口定义，就可以被方便地集成到分布式系统中。

2.2.4　软件体系结构

软件体系结构又称软件构件结构 SCI（Software Component Infrastructure）或软件构架，是对软件系统整体结构的刻画。软件体系结构是软件工程继过程性模型和面向对象模型之后的最新模型。通过对象总线，这种模型使跨越不同异构环境上的应用程序的开发成为可能。同时，它们通过引入框架使软件工程从工程化编程及类库再前进了一步。框架为构件提供结构，构件是重用的现成软件部件，构件和框架通过对象总线与别的构件和框架连接。也就是说，基于构件开发 CBD（Component Based Development）以软件体系结构为组装蓝图，以软件构件

为组装预制块，支持组装式软件复用，是提高软件生产效率和产品质量的有效途径之一，它提供了构件结构在最高层次上的代码重用。对象、构件、框架和对象总线组成了分布式软件应用程序。

软件体系结构研究如何应用可复用构件系统快速可靠地对系统进行构造，着重于软件系统自身的整体结构和构件间的互连与通信。主要包括：软件体系结构原理和风格，软件体系结构的描述和规范，特定领域软件体系结构，软件构件的集成机制等。

软件体系结构描述的是系统整体设计格局，它为 CBD 提供了构件组装的基础和环境。研究软件体系结构有利于发现不同系统的高层共性，保证灵活和正确的系统设计，对系统的整体结构和全局属性进行分析、验证和管理。将框架作为构造和演化的基础，可以实现大规模、系统化的软件复用。

软件体系结构是关于软件在系统级层次上的组成和行为，是设计过程中不可缺少的一个阶段，它对复杂软件的后期设计活动起到了决定性的作用。

虽然软件体系结构已经在软件工程领域中有着广泛的应用，但迄今为止还没有一个被大家所公认的定义。许多专家学者从不同角度和不同侧面对软件体系结构进行了刻画，较为典型的定义有：

① Dewayne Perry 和 Alex Wolf 的定义：软件体系结构是具有一定形式的结构化元素，即构件的集合，包括处理构件、数据构件和连接构件。处理构件负责对数据进行加工，数据构件是被加工的信息，连接构件把体系结构的不同部分连接并组合起来。这一定义注重区分处理构件、数据构件和连接构件，这一方法在其他的定义和方法中基本上得到了保持。他俩于 1995 年在 IEEE 软件工程学报上又采用如下的定义：软件体系结构描述的是一个系统各构件的结构、构件之间的相互关系以及对系统进行设计的原则和随时间进化的指导方针。

② Mary Shaw 和 David Garlan 认为，软件体系结构是软件设计过程中的一个层次，这一层次超越计算过程中的算法设计和数据结构设计。体系结构问题包括总体组织和全局控制、通信协议、同步、数据存取，给设计元素分配特定功能，设计元素的组织、规模和性能，在各设计方案间进行选择等。软件体系结构处理算法与数据结构之上的关于整体系统结构设计和描述方面的一些问题，如全局组织和全局控制结构，关于通信、同步与数据存取的协议，设计构件功能定义，物理分布与合成，设计方案的选择、评估与实现等。

③ Kruchten 指出，软件体系结构有 4 个角度：概念角度、模块角度、运行角度和代码角度。概念角度描述系统的主要构件及它们之间的关系；模块角度包含功能分解与层次结构；运行角度描述了一个系统的动态结构；代码角度描述了各种代码和库函数在开发环境中的组织。

④ Hayes Roth 则认为，软件体系结构是一个抽象的系统规范，主要包括用

其行为来描述的功能构件和构件之间的相互连接、接口和关系。

⑤ Barry Boehm 和他的学生提出，一个软件体系结构包括一个软件和系统构件，互连及约束的集合；一个系统需求说明的集合；一个基本原理用以说明这一构件，互连和约束能够满足系统需求。

⑥ 1997 年，Bass，Clements 和 Kazman 在《使用软件体系结构》一书中给出如下的定义：一个程序或计算机系统的软件体系结构包括一个或一组软件构件、软件构件的外部的可见特性及其相互关系。其中，“软件构件的外部的可见特性”是指软件构件提供的服务、性能、特性、错误处理、共享资源使用等。

⑦ 国内专家对软件体系结构的定义是，为软件系统提供了一个结构、行为和属性的高级抽象，由构成系统的元素的描述、这些元素的相互作用、指导元素集成的模式以及这些模式的约束组成。软件体系结构不仅指定了系统的组织结构和拓扑结构，并且显示了系统需求和构成系统的元素之间的对应关系，提供了一些设计决策的基本原理。这种定义与 UML 对软件体系结构描述的理念相类似。

总之，软件体系结构的研究正在发展，软件体系结构的定义也必然随之完善。目前，对软件构架的共识如下。

(1) 构件化

将系统中的部分软件设计成为易于复用的构件。

(2) 层次化

根据软件功能的不同，将不同的运行单位（构件）划分为不同的层次。典型的划分为：接口层，负责与用户的交互；应用服务层，又称业务逻辑层，负责完成具体的业务；数据服务层，以数据库为中心，负责保存相关的数据。

(3) 标准化

指连接软件系统中各部件的基础结构的标准化，实际上也是构件间互操作的标准化（如 CORBA，DCOM 等）。互操作标准的建立有效地分离了相互连接的构件，消除实现语言和物理分布等方面的障碍，为构造大规模的软件系统提供了基础。

软件体系结构表示了一个软件系统的高层结构，主要特点有：

① 软件系统结构是一个高层次上的抽象，它并不涉及具体的系统结构，比如 B/S 还是 C/S，也不关心具体的实现；

② 软件体系结构必须支持系统所要求的功能，在设计软件体系结构的时候，必须考虑系统的动态行为；

③ 在设计软件体系结构的时候，必须考虑现有系统的兼容性、安全性和可靠性，同时还要考虑系统以后的扩展性和伸缩性。所以有时候必须在多个不同方向的目标中进行决策。

2.3 基于构件和框架的软件开发

我们可以用构件、框架和对象总线来构架应用程序。这是一种全新的软件工程模式，如图 2-2 所示。

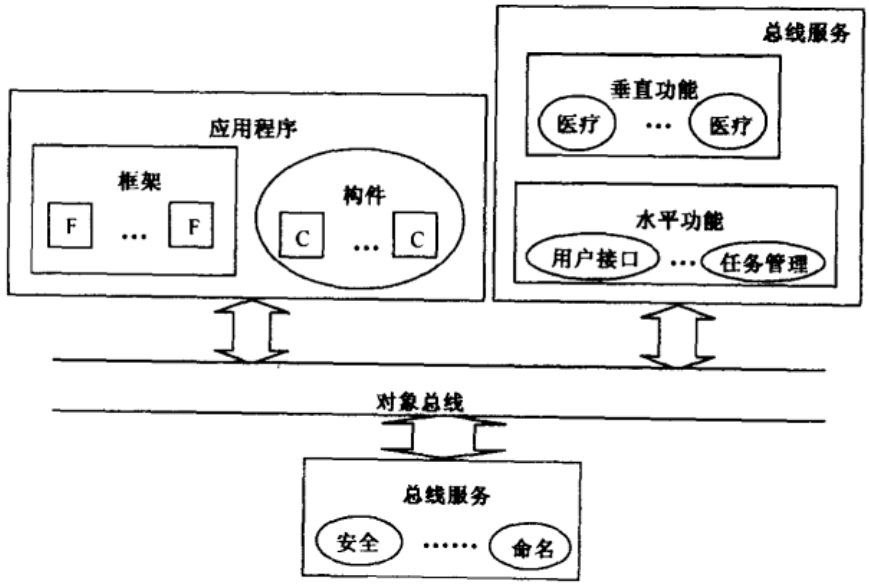


图 2-2 基于构件、框架和对象总线的应用程序模型

与一个对象总线进行通信的对象就不可能与别的对象总线发生关系，其原因是：

① 对象用来向总线描述自己接口定义的语言可能与别的对象总线的语言不同；

② 对象需要有一个框架来让对象总线处理远程调用中的某些细节，这种框架可能是该对象总线所特有的。

与传统的软件开发不同，在基于构件和框架的软件工程中，应首先进行软件框架的实例化，产生特定应用领域中的软件框架蓝图；之后进行构件的规划、设计和开发。如何将传统的软件开发方法与构件对象和框架合理地结合在一起，至今还处于发展和完善之中。一般而言，基于构件和框架的软件开发过程包括如下步骤：

① 设计。包括软件框架蓝图的规划，在抽象层描述系统中包含的构件，以及构件的接口、属性等。

② 部署。根据软件框架环境，决定构件的分布、构件之间的关系和实现细

节等。

③ 具体化构件。将逻辑构件转化为物理构件，使构件与具体的代码对应起来。

④ 与框架的具体化组合。将构件与框架的逻辑连接转化为物理连接，并以代码的形式表现出来。

⑤ 编译连接或动态配置产生最终目标码。

对构件和框架进行复用的目的在于，使用目前可获得的构件和框架构造出可运行的应用程序或者更大的构件。构件组装过程是：利用现有的构件加入到一个框架中，然后将所加入的构件利用事件进行连接构成。在构件之间通过框架建立起连接是构件组装的重要过程。在框架理论的支持下，分布构件间的连接是在不改变目标码的前提下基于框架的组装，虽然其连接机制比较复杂，但构件间的连接方式就变得非常灵活，尤其是增加了动态配置的功能，使用户可以在系统运行过程中增加新的构件或替代它的构件，并且不影响系统的正常运行。

参 考 文 献

彭德纯等．1995．分布式并行技术导论．武汉：武汉大学出版社

王柏等．2000．分布式计算环境．北京：北京邮电大学出版社

朱海滨，蔡开裕等．1997．分布式系统原理与设计．长沙：国防科技大学出版社

第3章　中间件技术

目前，分布式计算技术得到了非常广泛的应用，这是因为首先分布式计算的互连性有助于改善软件的互操作性；其次，通过采用并行处理技术提高了软件的性能，并通过副本技术提高其可靠性和有效性；另外，由于采用了模块化技术，分布式计算还显著改善了软件的收缩性和可移植性；同时，分布式计算的资源共享与开放系统特性提高了系统的性价比。但是，开发一个分布式应用软件，并使其各部分能可靠有效地协调工作则是一项困难的工作。原因有三：其一，由于开发分布式应用软件的常规工具和技术自身的局限性，使分布式应用的开发复杂化；其二，分布式应用的开发大量采用了功能分解技术，而当前常规的面向功能编程技术开发应用软件时往往会导致所生成的系统结构缺乏可扩展性，从而进一步增加了应用软件开发的复杂性；其三，由于工程上的考虑和对原系统的继承问题，使分布式应用大多基于异种平台，而如何将这些异种环境集成在一起牵涉到许多复杂的技术手段，显得困难重重。

目前，中间件技术的发展来源于以下几个应用需求驱动点：

① 消息驱动（message driven）：基于统一消息表示，采用点到点或消息代理集成结构，实现数据资源共享；

② 应用驱动（application driven）：基于基础通信中间件和构件管理平台，实现应用连接；

③ 流程驱动（process driven）：支持企业流程再造，加速客户、供应商、合作伙伴和员工之间的动态电子商务进程；

④ 用户驱动（user driven）：通过统一的界面访问所需要的任何信息，并控制应用的运行，从而加强协作，实现系统功能的快速扩展；

⑤ 模型驱动（model driven）：为企业应用开发和管理人员提供可视化的布局和设计能力，为开发者建立、发布和管理集成的应用和服务提供全面的支撑；

⑥ 知识驱动（knowledge driven）：基于商业智能技术，结合企业数据资源、业务逻辑和业务流程，解决企业的信息过剩问题导致的决策难题。

在分布式应用软件的开发过程中，中间件技术得到了更为广泛的重视，因为中间件所提供的平台透明性、通信协议透明性、硬件无关性，可以有效地降低分布式软件开发的复杂性及成本，提高软件的复用率。

3.1　中间件的概念

3.1.1　计算模式的发展过程

迄今为止，网络计算机模式的发展经历了 3 个阶段：以大型机为中心的计算模式、以服务器为中心的计算模式和客户机/服务器（Client/Server）计算模式。

以大型机为中心的计算模式称为分时共享模式，它是采用大型机作为主机并配备多个终端组成一个系统。这种模式是利用主机的能力，主机是系统的核心，一旦主机出了故障，整个系统便瘫痪。

以服务器为核心的计算模式则是通过网络将多台计算机相连，以实现资源共享，故此模式亦称为资源共享模式。这一模式是利用各站点的能力对所有应用进行运行，用服务器的能力作为外设的延伸。

客户机/服务器模式由客户机、服务器和连接部件 3 部分组成。在此模式下，应用被分为前端（客户端）和后端（服务端）两部分。客户部分运行在微机或工作站上，而服务器部分则可运行在微机或大型机的各种计算机上。虽然客户机和服务器工作在不同的逻辑实体中，但它们却能协同工作。客户机/服务器模式的最大特点是系统使用客户机和服务器的智能、资源和计算能力来共同执行一个特定的任务。

3.1.2　C/S 结构模型与中间件

在 C/S 结构中，所有网络节点可划分为客户机和服务器两类。C/S 结构由 4 部分组成，如图 3-1 所示。

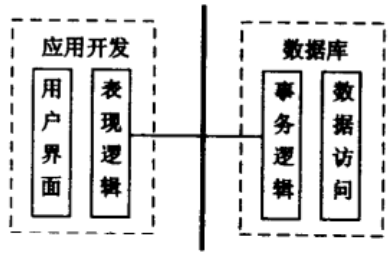


图 3-1　C/S 结构模型

客户端通常包括用户界面和表现逻辑，其中，用户界面主要完成应用的前端处理及末端交互，诸如屏幕格式显示、键盘处理等，它不能直接访问数据库中的数据；表现逻辑则向各种应用提供一个统一的数据库访问接口。

服务器端通常包括事务逻辑和数据访问，其中事务逻辑主要完成数据的安全性及事务的完整性等管理工作，它可跨越多个服务器协同操作。

C/S 应用开发与一般应用开发的不同之处是将事务逻辑放在服务器上，这样能够提高应用的可靠性，并减少系统开发和维护的工作量。

C/S 模式的主要特点是分布处理，以及客户机与服务器之间信息交换量的大大降低。但随着信息技术的发展，这种结构的缺陷也被逐渐地暴露了出来。首先，进入 20 世纪 90 年代中期，信息技术迅猛发展，CPU 的处理能力越来越强，网络的应用日益普及，软件应用的规模和范围无限扩展，许多应用程序需在网络环境的异构平台上运行。在这种异构的环境下，不但存在多种硬件系统平台，而且包括运行在这些硬件平台上的各种系统软件及风格不同的用户界面，这些硬件系统平台还可能采用不同的网络协议和网络体系结构连接。由此带来的问题也越来越明显，如不同硬件平台、不同网络环境、不同数据库之间的互操作问题，多种应用模式并存、传输不可靠、数据加密问题，开发周期过长等等。单纯依靠传统的系统软件或工具软件提供的功能已无法满足需要。其次，当 C/S 方式逐渐推广到企业级的关键任务环境时，便出现了一些问题，如系统可扩展性差、解析度低、可管理性差、维护代价高、安全性差、系统通信功能较弱等。

为了解决两层结构遇到的问题，人们提出了 3 层结构的概念，即在传统 C/S 体系结构中，加入一个中间层，使其扩展成客户机—中间层—服务器 3 层结构，以解决不同系统之间的互操作问题。

3 层结构就是在原来的两层结构的客户端和数据库服务器端之间增加了一个中间层：应用服务器层，使业务逻辑从客户端移到了应用服务器端。这样一来，客户端仅负责显示用户界面和处理用户的输入/输出，就不再和数据库建立连接了。客户端把用户的请求发送到应用服务器，由应用服务器从数据库服务器获得数据并进行计算，计算结果返回该客户端进行显示。应用程序集中放置在中间层上，由所有用户共享。当事务逻辑发生变化时，只需更新服务器上响应的应用程序构件，所有的客户端就可以使用新的事务处理逻辑，避免了客户端应用程序版本控制和更新的困难。

3 层 C/S 方式以中间层管理大量的客户端并为其连接、集成多种异构的服务器平台，通过有效的组织和管理，在极为广阔的范围内将客户机和服务器进行高效的组合，并以其可扩展性、安全性、易维护性等方面的优势，在企业级的应用中显示了强大的生命力。

但是，即使采用上述的 3 层 C/S 结构，对于不同开发工具开发的应用程序，也可能要安装不同的客户端软件。对于一些在 Internet 上开办的企业，可采用基于 Web 的应用结构。基于 Web 的应用结构实际上是 3 层结构中的一种特殊情况，即在客户端和服务器端之间加入了 Web 服务器，从而使系统在逻辑上分为用户层、Web 服务器层（包含应用层）和数据层。另外，基于 Web 应用中的应用服务器大都采用基于对象构件模型的中间件进行开发，如 BEA 的 WebLogic、

IBM 的 Websphere 等。根据信息专家 Standish Group 的调查报告显示，采用一个成熟的中间件产品能够为应用开发节约 30%～50%的时间。

随着信息技术的迅速发展，尤其是 Internet 和 WWW 的出现及其在数字城市乃至数字地球中的应用，必然使软件的开发和使用面对网络这样的异构环境。在这种分布式的异构环境中，就会有多种现象出现：存在不同的硬件；存在不同的操作系统；存在不同的系统软件、应用软件和开发工具；存在多种风格各异的用户界面；存在不同的网络连接等等。所有这些问题，对软件的开发提出了新的挑战，尤其是对大规模软件的开发。如何在这样的异构环境中充分利用资源并开发新的应用，是一个非常现实而困难的问题。

传统的程序设计语言是面向过程模式的，进入 20 世纪 90 年代，研究的中心转移到了面向对象的模式。与此对应，开发者开始创建中间件系统，该系统把过程调用扩展到远程调用，进而扩展到了计算机之间的方法调用。由此可见，中间件是为解决分布异构问题而提出的，中间件的使用使在异构环境中进行开发和应用变得更为方便和容易。

以上是从计算模式的发展方面阐明了中间件的作用。再者，从目前的国内外状况来看，中间件已经同操作系统、数据库管理系统并驾齐驱，成为基础软件的 3 架马车之一，特别是近几年，只要是分布式应用软件几乎都是在中间件的基础上开发的。

3.1.3　中间件的定义

中间件正处于发展过程中，给出中间件的确切定义是非常困难的。但由于中间件位于硬件、操作系统和应用程序之间，能满足大量应用需要，运行于多种硬件和操作平台之上，支持分布计算，提供跨网络、硬件和操作系统平台的透明性的应用和服务，支持标准的接口和协议。所以一般认为，中间件是泛指能够屏蔽操作系统和网络差异，为异构环境之间提供通信服务的软件，是具有强大通信能力和良好可扩展性的分布式软件管理框架。简而言之，中间件是位于平台和应用之间的具有标准接口和协议的通用服务。

对于应用软件开发来说，中间件远比操作系统和网络服务更为重要。中间件提供的程序接口定义了一个相对稳定的高层应用环境，不管底层计算机硬件和系统软件如何更新换代，只要将中间件更新，并保持中间件对外的接口不变，应用软件几乎不需任何改动，从而保持了企业应用软件开发和维护中的重大投资。

3.2 中间件的功能、特点和分类

3.2.1 中间件的功能

中间件应具有如下功能：

① 中间件能够全面支持各种事务。包括支持本地事务、远程事务以及分布于多个异地的访问多个资源管理器的分布式事务。能够协调多个资源管理器进行事务处理并通过两段提交协议来保证事务的完整性。

② 对大型数据库管理系统的支持。中间件能够支持多种数据库管理系统，它与数据库管理系统协同工作以有效地对事务进行处理。

③ 支持客户机/服务器交互。客户和服务器进程可以在相同和不同的节点上，中间件能够支持客户对服务器进程的透明访问。并且能够自动地产生和管理应用程序的多个拷贝，并执行必要的负载平衡。在具体实现中，可将消息传递机制和远程过程调用机制进行结合，以获得良好的性能。

④ 对远程客户的支持。中间件允许一个使用了中间件接口的客户应用程序，在一个没有运行该中间件内核的远程工作站上运行，这可为开发者和用户带来方便。一是负载的均衡，占用 CPU 资源较多的客户应用程序可以脱离中间件系统节点而运行，以腾出更多的系统资源给其他的任务；二是可充分发挥客户平台的优势和资源，如运行于 PC 机上的 Windows 客户应用程序就能利用 Windows 所提供的图形用户界面等。

⑤ 对应用程序的支持。中间件应为应用程序提供许多服务，如分布式应用程序的 C 语言开发工具、分布式调试工具等。

⑥ 对大量用户的支持。中间件能够支持大量的用户。它通过对系统资源的有效控制来增加更多的用户数。

⑦ 对应用执行方式及性能的优化。中间件应能为应用部件设置不同的选项，以便管理员能够在系统的牢固性和系统性能之间进行协调和折中，以便通过应用程序的实例数对系统进行控制来优化系统的吞吐率。

此外，中间件还应该具有能在多种平台上运行、支持多种协议以及对国际化的支持等功能。

总而言之，中间件应能够屏蔽硬件和网络，提供并优化通信机制，使服务器位置透明，提供应用的可扩展性，保证交易和数据的一致性，保证信息的可靠传输，提供应用的安全机制等，在分布式的客户机和服务器之间起到了承上启下的作用。

3.2.2　中间件的特点

中间件具有如下一些特点。

(1) 易集成性

中间件能无缝地接入应用开发环境中，应用程序可以很容易地定位和共享中间件提供的各种应用服务。

(2) 移植性

对于应用程序来说，中间件将与平台有关的细节隐藏起来。因此可以在不改变应用程序代码的情况下改换计算机底层硬件、操作系统或通信协议。

(3) 分布透明性

分布透明性是中间件的重要组成部分，它屏蔽了由于系统的分布所带来的复杂性。在中间件中主要包括如下 8 个方面的内容：

① 访问透明性，能屏蔽数据表示和调用机制的差别，使得不同对象之间可以协作；

② 位置透明性，能屏蔽分布式环境中有关对象的物理位置的信息，好像是在本地使用该对象；

③ 迁移透明性，系统已经把一个服务对象迁移到另外的位置，而不影响对该服务对象的使用；

④ 失败透明性，能屏蔽对象的失败和可能的恢复，使系统具有容错性能；

⑤ 重定位透明性，能屏蔽交互中的一个对象已经被搬家；

⑥ 复制透明性，能屏蔽某一对象已经在分布式系统的几个位置上被复制；

⑦ 持久透明性，能屏蔽系统中对象的激活、撤消和再激活；

⑧ 事务处理透明性，能屏蔽为获得相应对象之间的完整性所提供的机制。

(4) 易演进性

由于中间件实现的功能对应用程序来说是透明的，所以可以对局部进行改进而不会影响到系统的其他部分。

(5) 高可靠性

中间件提供了接管和恢复功能，保证事务及关键性业务不被丢失，如银行事务、交易事务等。

(6) 易使用性

中间件能和同构或异构环境下的多种数据源通信，同时它能管理数据间的公共逻辑约束。它将用户从复杂的平台、网络、数据库选择中解放出来。

(7) 高可用性

中间件提供了多个功能来提高大型应用程序的可用性。

(8) 健壮性

可以通过本地和全局两级系统管理来动态地启动和停止某一部件的运行，动态地诊断运行部件的错误，动态地连接新的节点。也就是说，它能够迅速地检测出应用程序和中间件系统内部失败，对于失败的应用程序，将被自动的复制以保证其可用性，而无需关掉整个系统。

3.2.3 中间件的分类

中间件通常分为以下5类。

(1) 远程过程调用中间件

远程过程调用中间件RPCM（Remote Procedure Call Middleware）是使客户端的应用调用一个位于远端平台的进程或服务。目前广泛使用的RPC中间件有：

① ONC RPC：开放网络计算远程过程调用（Open Network Computing Remote Procedure Call），也称SUN RPC，它是由SUN设计的。ONC RPC定义了接口描述语言、客户和服务器之间的桩（Stub）程序之间通信的消息格式、外部数据表示标准，采用TCP/IP协议，并允许程序员选择UDP或TCP传输协议。

② DEC RPC：分布式计算环境（Distributed Computing Environment）是由开放组（The Open Group）定义的，它由许多协同工作的构件和工具组成，包括DEC/RPC及自己的接口描述语言（IDL）。DEC/RPC还允许一个客户访问多个服务器；任何情况下，DEC/RPC都可以使用TCP/IP协议，并允许程序员选择UDP或TCP传输服务。

③ MS RPC：微软远程过程调用（Microsoft Remote Procedure Call）源于DEC/RPC。它与DEC/RPC有相同的核心概念和总体程序结构，但在一些细节上保持了自己的特性，如有自己的IDL、客户和服务器桩程序之间的通信协议等。为了更好地支持对象，微软开发了新一代MS RPC——MS RPC2。

此外，还有OSF的RPC，Netwise RPC，IBM的OS/2分布式连接（DPL）和SYBASE的RPC等。

在这些众多的RPC中，DEC的RPC是最为先进的，它包括针对网络传输独立性和透明性的特殊语义，采用POSIX标准异步线程，支持并发和并行处理。

(2) 消息中间件

消息中间件MOM（Message Oriented Middleware）利用高效可靠的消息传递机制实现异构平台之间的通信。

(3) 数据库访问中间件

数据库访问中间件DAM（Database Access Midddleware）实现对来自不同厂商数据库的访问。它提供了一系列应用程序接口（API），通过中间件而不考虑操作系统及网络来访问数据库。ODBC、JDBC都是数据库中间件的标准。

(4) 对象中间件

对象中间件也称对象请求代理 ORB（Object Request Brokers）。ORB 提供一种通信机制，透明地在异构的通信环境中传递对象请求，这些对象可以位于本地或远程机器，且对象之间的客户机/服务器的角色是可以互换的。

（5）交易中间件

交易中间件也称分布式事务处理中间件 DTPM（Distributed Transaction Processing Middleware），它提供了重要的分布式应用服务。通过事务处理监控有效的平衡网络负载和主机服务，从而减轻了阻塞，提高了系统性能。

3.2.4　中间件的发展趋势

中间件的发展趋势是向 Internet 的延伸。Internet/Intranet 技术早已在全球范围内广泛采用，但由于其自身的技术特点，在构造许多大型企业级应用时仍显得不足，如并发控制、负载均衡、可靠传输、数据路由等，因而仍然存在供中间件发展的中间地带。基于对象构件模型的中间件（也称对象-构件请求代理）是中间件一个发展迅速的分支，以其跨平台和对 Java 的支持很快在 Internet 应用中占据主导地位。尤其是基于 EJB 的对象中间件，有专家称之为代表着中间件的未来。

由于目前大型企业纷纷构筑自己的 Internet 应用，以银行、铁路定票为代表的交易型业务仍然很多，而可靠的消息传递也是每一个跨区域企业级业务所不可缺少的需要，这就要求中间件能够向集成和综合发展，将各种不同用途的中间件集成在一起，给企业提供一个完整的解决方案。

3.3　中间件基本框架模型和工作机理

3.3.1　中间件基本框架

中间件按功能特点分为 4 层：传输协议层、应用平台层、中间件服务层和应用接口层。中间件的基本功能框架结构如图 3-2 所示。

（1）传输协议层

由各种传输协议组成，实现各种机型、各种网络之间可靠的通信。如 TCP/IP、DPX/SPX、NetBIOS 等。

（2）应用平台层

列出了中间件运行的各种支撑平台，如 Unix、Windows NT、NetWare、OPENVMS 等，实现应用的跨平台操作。

（3）中间件服务层

提供了各种标准服务，如通信、控制、计算和管理等。

① 通信服务。使得一个应用服务能和其他本地或远程的应用进行通信。包

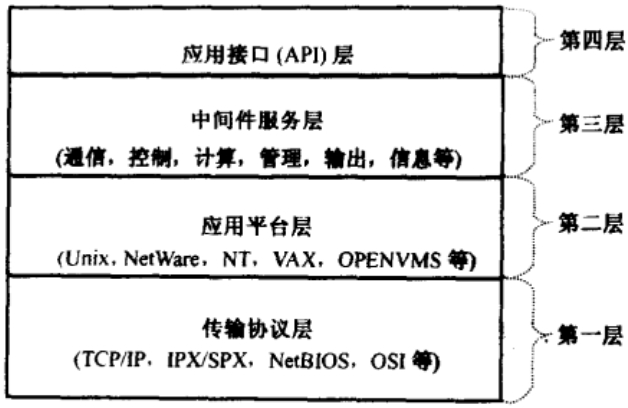


图 3-2 中间件的基本功能框架结构

括远程过程调用服务、消息队列服务、传输服务（邮件信息传输、电子数据交换）等。

② 控制服务。能使应用程序在本地或分布环境中控制程序的执行。包括连续计算服务、多线程服务、目标代理服务和事务处理监控服务等。

③ 计算服务。提供了一系列使应用程序能够进行复杂计算的性能。提供了国际化的应用，能使用不同的字符集，具有对分布在不同时区的时间进行同步和管理的功能等。

④ 管理服务。包括网络管理、系统管理、安全管理等。

⑤ 输出服务。主要功能是显示信息和用户进行交互。包括图形服务、打印服务、终端服务等。

⑥ 信息服务。使应用程序能定义、存储、存取和管理数据，支持多存取机制。包括数据存取、文件存取、目录存取等。

(4) 应用接口层

定义了应用程序怎样和中间件进行交互，包括编程语言、系统环境和交互机制的定义等。

3.3.2 中间件工作机理

从理论上讲，中间件具有以下工作机理：客户端上的应用程序需要从网络上的某个地方获取一定的数据或服务，这些数据或服务可能处于一个不同硬件环境和运行着不同操作系统及特定查询语言数据库的服务器中。客户机应用程序中负责寻找数据或服务的部分只需访问一个中间件系统，由中间件完成在网络中找到数据源或服务，进而传输客户请求到服务器，并把来自服务器的答复信息经过重新组合，最后将结果送回到客户端的应用程序。

一般情况下，中间件支持下的应用程序从逻辑上可划分为两个部分，一部分负责程序的主体，另一部分负责访问中间件。这种逻辑上的划分十分有利于分布式 C/S 环境下的程序编写，无论数据或服务处于多少个运行着不同操作系统的主机上，开发者不用编写传输层指令到每个应用程序中，只需在其软件中写入一些简单的指令，来调用中间件提供的 API 函数。当发布应用程序投入使用时，中间件会迅速、有效地在应用程序和各种操作系统、通信协议或数据库系统之间建立起一道桥梁，使 C/S 环境发挥出最佳效能。中间件的工作机理如图 3-3 所示。

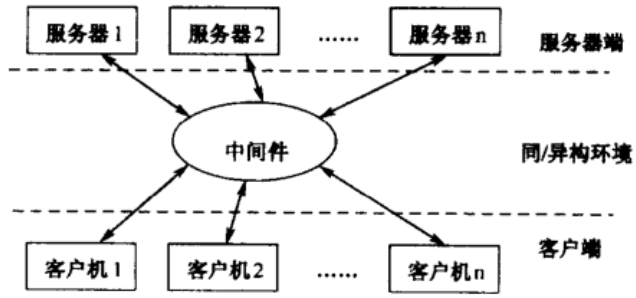


图 3-3　中间件的工作机理

3.3.3　Web 环境中的中间件

中间件的产生和发展与 C/S 结构的发展是紧密结合的，正是由于 C/S 环境一直存在着操作系统、文件格式、网络协议、服务等相互异构的多元化问题，才使得中间件作为不同节点间协同工作的桥梁得以不断发展。随着 Web 技术的飞速发展，相对统一的浏览器界面和跨 Internet/Intranet 的超文本传输使得异构系统的节点不在面临以往中间件所解决的那么多难题。所以，中间件和 Web 技术是紧密结合的，更多的工作则转向数据库服务和 Web 服务之间的连接之上。工作机制如图 3-4 所示。

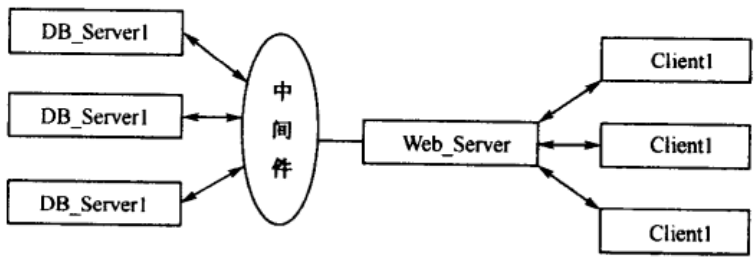


图 3-4　Web 环境中的中间件

3.4 中间件实现的关键技术

中间件的实现涉及到许多技术难点，但其关键技术是远程过程调用 RPC 和报文服务。

1. 远程过程调用 RPC

通过 RPC 机制，一个进程可以执行另一个驻留在其他系统中的进程。

透明性完全屏蔽了 RPC 传送机制的细节，一个专门的 RPC 工具可支持一种或多种不同的传输机制。一个成功的 RPC 实现要求调用者能迅速有效地找到被调用例程所在的服务器，而完成此搜索的一种方法是建立从子例程到服务器的映射数据库。各映射数据库中的所有条目可进行修改，以便动态地分配目标服务器，达到 RPC 的一次调用到另一次调用的变更。在使用中，被调用的子例程名要与映射数据库中的名字相匹配，一旦找到服务器，RPC 使客户机和服务器相互认证，检查它们的安全特权级别。RPC 还允许用户采用面向对象的程序设计方法。故 RPC 与其集成服务一起构成了一个中间件。它在运行时表现出如下特点：

① 对基础网络协议的透明性和独立性；
② 支持可靠传递、检错检测以及网络故障恢复；
③ 既能支持网络寻址，又能支持目标服务；
④ 支持并行和并发处理的多线程；
⑤ 支持异构环境中的可移植性和互操作性；
⑥ 支持资源的整体一致性和应用安全性。

特别值得一提的是，数据库 RPC 是目前应用比较广泛的 RPC 之一，它是由某一客户或另一服务器通过网络发送一个 DBMS 的请求。相对于传统的 RPC 来说，数据库 RPC 不仅能调用存储过程，而且对单个请求允许 DBMS 服务器返回多条记录，以降低网络负载。同时，由于 RPC 解决了语言差异问题，也帮助实现了各种 DBMS 之间的连接问题。

2. 发报文和排队

由于许多 RPC 采用的是同步机制，所以使其在 C/S 下的适应性相对变差。RPC 的同步机制要求客户必须等待服务器完成为止，这在许多 C/S 应用中是用户所不希望的。而发报文是解决此类问题的有效方法。

发报文通常是一种无等待的通信技术，即各通信伙伴并不相互等待以交换报文，也不管可供使用性和访问性。发报文实质上是使数据和控制分布的过程，报

文不仅可以表示成数据和控制信息，而且还可以表示成数据分组、SQL 串、图形图像和声音等。排队则是一种无连接的通信技术，它允许诸通信实体保存信息，直到预计中接收者准备接收为止。故排队中各实体可按各自速度操作，不需同步。异步报文排队技术主要有如下特点：

① 时间、基础网络和通信机理的独立性；

② 结构的灵活性；

③ 队列和应用管理的分开性；

④ 较低的网络连接数；

⑤ 信息流格式的灵活性。

RPC 和发报文都是提供一种透明而一致的程序，通过程序的通信来满足许多中间件的要求。但两种形式各有优缺点：RPC 的主要优点是以标准为基础，不需要新 API；发报文的主要优点在于支持同步和异步的报文传递和处理，无等待连接，不需要预编译程序。

由此可见，实现中间件有效性的关键就是如何将发报文机制和排队机制结合起来，形成一种混合机制，以供实际使用。

3.5　五大类中间件的工作机理

按照中间件的不同用途，将中间件分为 5 大类：远程过程调用中间件 RPCM、消息中间件 MOM、数据库访问中间件 DAM、对象中间件和交易中间件 DTPM。下面对它们的工作机理分别加以阐述。

3.5.1　远程过程调用中间件

远程过程调用 RPC 是从一台机器或一个进程调用另一台机器或另一进程的服务或方法，这种调用是通过网络来实现的。远程过程调用是创建分布式应用的一种方法，来源于 Unix 操作系统的处理思想，被普遍认为是创建分布式应用的有效方法。

远程过程调用早已不是一个新的概念了，从程序员的角度来看，RPC 采用常规的编程模式：程序代码调用远端过程并将结果返回。这种编程模式是程序员最为熟悉和最易理解的模式。当使用 RPC 时，只需编写很少的网络程序代码，绝大部分代码由 IDL（Interface Define Language）生成。

远程过程调用的机制框架可用图 3-5 所示。首先是对远程过程调用的顺序的定义，即用中间件中的 IDL 对调用顺序加以描述；然后是客户端和服务器端通信的管道生成，即利用 IDL 编译器对这种定义和描述进行编译，从而生成支持客户机和服务器进行通信的管道。在服务器端，被调用的远端过程需单独编写，

并被置入服务器端 Stub（即 Skeleton），同时要在远程过程调用中间件中进行注册，以备调用。在远端过程调用运行时，首先启动的是本地客户端的 Stub，然后由 Stub 捆绑需要调用的远程过程名称和参数，并将这些通过网络传递给服务器端的 Stub，实现不同平台之间的数据格式的转换和参数的传递。相反的过程则将调用结果由服务器返回到客户机。这实际上是一种调用/返回（call/return）模式。

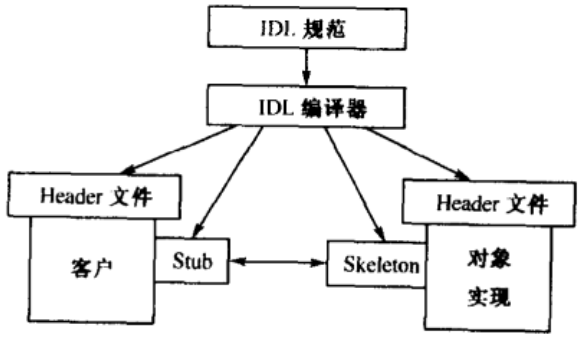


图 3-5 IDL 编译器结构

RPC 一般采用调用/返回模式（图 3-6），多用于应用程序之间的通信，而且采用同步方式。RPC 程序之间的同步通信一般采用 Request-Wait-Replay 方式。因此，RPC 更适应于小型简单而不需要采用异步通信方式的应用，而对大型复杂场合的应用，因为它需要程序员考虑网络或系统的故障、处理多个网络的连接、可移植性、缓冲及流量控制和进程之间的同步等多种问题，所以不太适应。

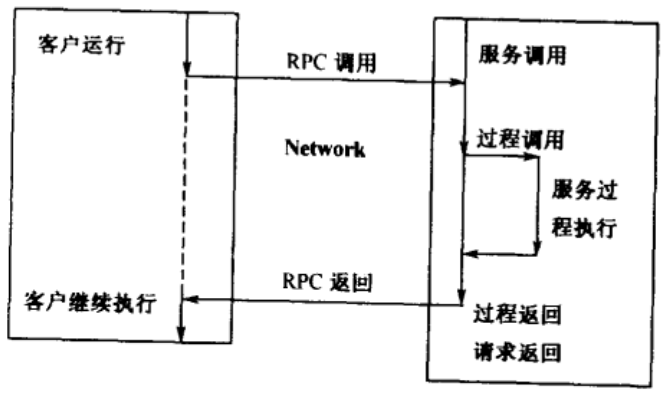


图 3-6 RPC 工作流程

远程过程调用的特点是：

① 客户/服务器模式。远程调用过程实际上是通过客户/服务器方式实现的，

因为其实现模型就是客户端的应用调用一个位于远端服务器平台的进程或服务。

② 数据封装性。远端过程调用负责捆绑参数，实现不同平台之间参数的传递，发送参数到远端过程。

③ 同步性。即当客户机发出请求时，服务器进程处于运行状态。值得注意的是使用线程（thread）可以实现异步模式。

④ 复杂性。远程调用的实现环境可能需要系统级的任务来创建，因此较复杂。但是创建一旦完成，对程序员就是透明的了。

由此可见，远程过程调用的缺点是需要进行系统级的配置，优点是可以方便地实现不同平台之间的数据转换。

远程过程调用和消息中间件的相似之处是，通过激活一个远端进程来实现通信。由于实现机制的不同，远程过程调用更适合应用于客户/服务器计算方式；而面向消息的传输本身就支持异步传输，既可支持客户/服务器模式，也可支持对等计算方式，即在不同的情况下，调用和被调用的关系可以互换。

3.5.2　消息中间件

网络应用程序，经历了从简单到复杂，从集中到分布的演变。大型分布式应用程序的构建和维护却是一件十分复杂的事情，尤其是不同应用程序间的通信。大多数传统的通信技术要求发送和接收应用程序必须满足两个条件：一是它们同时在线；二是通过网络能同时通信，发送者和接收者需知道相互间程序的调用接口。

这就要求，一个应用程序一旦有任何接口变化，都必须通知其他应用程序。为了保持数据的完整性，发送和接收应用程序通常使用分布式业务，以确保任一方应用程序数据的改变能被双方承认。然而，实际情况往往是：

① 应用程序并不总是同时运行；

② 网络，尤其是广域网，并不总是可用的和可靠的；

③ 在所有者域对应用程序的改变，要求在其他域也作相应改变，这在技术上是不切实际的。

消息中间件 MOM 正是为了满足以上的需要而产生的。消息中间件并无标准可循，一般把工业标准 TCP/IP 协议作为基础。面向消息的中间件在 TCP/IP 网络体系结构中处于应用层（见图 3-7），网络应用程序（NAP）建立在面向消息的中间件之上，实现各种分布式应用服务。

MOM 还遵循 X/Open 的分布式事务处理模型，适应于分布式计算环境多样化、用户数目规模化、业务逻辑复杂化的发展要求，实现消息位置无关性、用户透明性。

消息中间件亦称消息队列中间件或通信中间件，它是依据消息传送或消息队

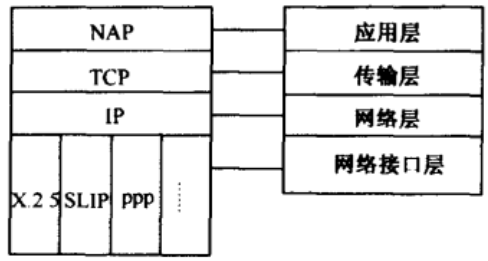


图 3-7 消息中间件在 TCP/IP 体系结构中的位置

列的原理来工作的。它能够简化应用之间数据的传输，提供可靠的、跨平台的消息传输手段。消息中间件支持同步和异步两种通信模式，其中异步通信模式是基于消息队列转发机制的。一般来说，消息队列广泛采用对等的分布式计算模型来实现分布式计算中的同步和异步交互。消息队列一般提供多协议支持、高端服务和其他系统管理服务，完成可靠的、可扩展的异构环境中的通信。

“消息”实质上是一个由用户定义的数据结构，一个消息可以包含一幅图像、若干个数据库更新记录、一个电子邮件等。每条消息由头信息和体信息组成。头信息是对消息结构的描述，对整条消息起控制作用，它含有消息的属性及其他与消息相关的系统信息，如消息标识、消息类型、目的队列名、日期时间等。其中的目的队列名，标识该条消息发往的目的队列，即接收该消息的队列。目的队列名可以表示成“队列名@队列管理器名”的形式，也由两部分组成，第一部分是用来标识目的中间件系统的队列管理器名，另一部分是标识目的消息队列的队列名。体信息主要是消息的应用数据，应用数据对于中间件来说是一串无意义的二进制字节串，是应用程序通信的数据，其具体的语义要由通信双方事先约定。由此可见，消息的内容中包含了在应用之间被传递的信息和数据。

消息的类型分为 3 种：

① 请求消息（request），用于一个进程向另一个进程请求发送数据，该消息要求对方一定要应答；

② 应答消息（reply），用于对请求消息进行响应；

③ 通知消息（report），单向而不需要应答消息。

消息中间件的工作原理是，应用之间以一系列消息的方式进行通信。在发送者和接收者的传送过程中，为了避免在传递过程中消息被丢失，消息保存在队列中。消息中间件为消息接收者查看消息提供了一个缓冲区域，应用把消息发送到与接收者相关的队列中，如果发送者想及时得到反馈，它们就把接收返回消息的队列名称包含在所有它们发送的消息中。消息传递机制要保证将发送者的消息传送到目的地。在消息传递中，应用程序之间不必建立直接的联系，也就是发送方

将消息放入队列中，然后接收方从自己的队列中提取消息。发送方在发送消息时不必关心接收方是否处于接收状态，这样可以保证消息传递的异步性。

由此可见，消息中间件的工作主要是通过将信息以消息的方式在程序间传递来完成。消息中间件一般可以采用两种形式：消息传递（message passing）和消息队列（message queuing）。消息传递常用于建立大型的分布式应用，其主要的模式是发布/订阅（publish/subscribe）方式。采用该方式，应用程序既可以订阅，也可以发布。发布/订阅是一种消息传递的常用形式，在这种形式中，应用对其感兴趣的主题进行登记，一旦一个主题被一个应用“登记”，那么这个应用就会接收到与该主题相关的消息。发布/订阅通信模型提供了位置透明性。程序只需要简单地将消息以主题方式发送出去，由中间件来负责将消息传递给所有订阅该主题的程序。MOM 主要通过 Agents 技术来实现发布/订阅方式的应用。当程序广播消息时，首先与一个代理进行连接，将消息传递给代理；代理负责路由消息给相应的程序。由于代理可以实现消息的动态路由功能，因此，该方式能够提供较好的容错性能，但它缺乏 MOM 的异步特性，不太适合长时间网络断开的情况。

消息队列方式允许程序无需直接建立起连接即可发送和接收消息。程序只需简单地将消息发送给消息队列，由消息队列负责消息的传递，对应用程序完全透明。消息队列采用异步模式，为信息提供了一个安全的存储方式，特别适用于不是直接连接的应用，如移动用户、发送方或接收方进程可能处于不活动状态的应用。它的缺点是需要一些配置工作，性能不是很高，而且如果队列丢失，整个系统将受到影响。

图 3-8 是消息中间件的体系结构。本地应用程序发送一条消息到消息队列接口，消息队列接口对消息队列进行重新组合，形成并一个包含消息路由信息的消息队列头，然后将其置于本地消息传输队列。消息通道代理利用传输协议和网络的物理连接将消息发送到远程系统。在另一端，对消息的读取刚好与消息的发送过程相反。远端的中间件读取消息路由信息，然后把消息置于相应的目的队列。远端的应用程序通过消息队列接口读取消息。

1. 队列管理器

队列是面向消息中间件的数据结构，用抽象数据类型来表示，包括属性和方法。队列中的属性有消息发送的次序（先进先出或带优先级的先进先出）、消息访问方式（共享或独占）、队列的长度和队列的触发机制等。队列的属性和行为由队列管理器进行管理。在队列中可以存放一条或多条消息。

队列管理器是面向消息中间件的核心部件，它负责创建和删除队列并控制队列的行为。其主要由消息路由模块、消息通道代理模块和系统管理模块组成。

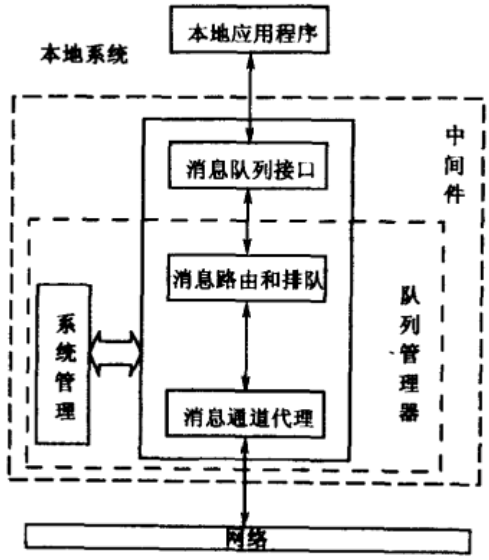


图 3-8 消息中间件的体系结构

消息路由和排队模块负责消息被传递的方向。消息路由算法如下：

① 中间件通过消息队列接口接收消息；

② 在中间件中，检查目的队列名中的队列管理器名是否是本地的，如果是，则将该消息置于对应于队列名的应用队列中，再转至⑤；如不是，则将该消息置于对应于队列管理器名的传输队列中；

③ 消息通道代理模块负责将该消息传递给远程中间件系统的队列管理器；

④ 远程中间件系统队列管理器将该消息置于对应于队列名的应用队列中；

⑤ 应用程序通过消息队列应用编程接口从应用队列中获取该消息。

消息通道代理（MCA）模块主要负责消息的底层传递和网络通道的故障恢复。对于前者，MCA 利用网络提供的通信机制，将消息投递给分布式系统中的底层传输协议（如 TCP 或 UDP 等）。多数情况下，通信双方采用握手式通信，即接收方的消息通道代理模块从目标消息队列读取消息之后，再向发送方的消息通道代理模块发送应答消息以确认消息已安全到达。对于后者，MCA 通过对网络通道的探测来获得网络中发生的故障，并负责故障的恢复。也就是说，当一个在队列管理器之间通过通道发送消息的操作失败时，MCA 暂时代为存储此消息。当网络连接再次建立后，MCA 自动重发这条消息，无需应用程序干预。

接收方 MCA 和发送方 MCA 之间除了传送消息之外，它们还通过记录和传送一些通道的状态来获得通道的故障。当一个通道在故障之后重新可用时，两端的 MCA 需要确定最后一个被正确传送的数据分组后，才能开始重新传递，以确保消息传递的可靠性和高效性。

系统管理模块使系统管理员能交互式地使用、维护和管理中间件，如建立和删除队列通道、生成和维护队列路由表、查询通道的状态和处理各种异常情况等。系统管理模块还负责在活动日志中记录所有不可丢失的消息和重要事件。当活动日志记录满后，系统管理模块将其拷贝到归档日志中。

2．触发机制

对于连续运行的应用程序，当消息流动量大且稳定时，可以及时读取任何到达的消息，但这毕竟是部分情况。在大多数情况下，消息的到达与应用程序的活动不总是一致的，这时就需要设置消息队列的触发机制来激活处于休眠状态的应用程序。图 3-9 是触发机制工作过程示意图，触发机制的工作过程是：

① 一条消息到达被触发的应用队列；

② 队列管理器根据环境信息，决定这条消息的到达是否构成一次事件触发的条件；

③ 当触发条件满足时，队列管理器创建一条触发消息，并将触发消息发送到启动队列；

④ 触发监控器从启动队列中读出触发消息；

⑤ 当触发监控器读取触发消息后，发出一条激活命令给触发队列相关联的应用程序；

⑥ 当应用程序被启动后，它从触发队列中取走消息并执行相应操作。

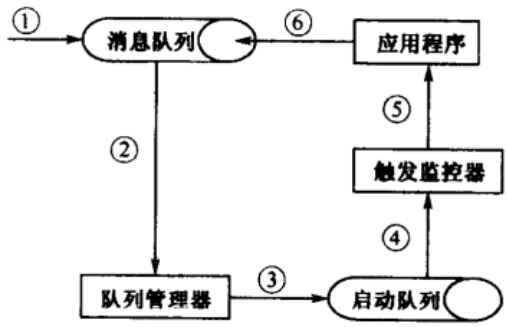


图 3-9　触发机制工作过程

触发消息中包含有被启动的应用程序与触发队列的关系，其数据结构如下：

```
struct app {
  char execName [32];
  char path [32];
  char parameter [32];
  char env [32]
```

```
    int copies;
    int ipcKey;
};
```

其中，execName 为应用程序的名称，path 为应用程序的工作路径，parameter 和 env 分别为启动应用程序时所需的参数和环境信息，copies 为被触发的应用程序的数量，ipcKey 为被触发的应用队列的键值。

消息触发类型有 3 种：每条消息触发、多条消息触发和有特定优先级的消息触发。

每条消息触发是任意一条消息到达队列时产生的，它适合于触发一次只处理一条消息的应用程序。每当有一条消息到达应用队列，应用程序都被触发。如果在短时间内有多条消息到达，则应用程序的多个拷贝被频繁激活，因而开销较大。这种触发适合于那些很少到达但十分重要、需要被立即处理的消息。

多条消息触发是队列上有特定数目的消息时产生的，它适合于一次处理多条消息的应用程序。当第 n 条消息到达应用队列时，队列中的 n 条消息被应用程序一并读取并加以处理。这种触发适合于能够批量处理的消息。

有特定优先级的消息触发适合于带优先级的先进先出消息队列，用来处理含有不同类型、不同层次、不同级别的多种消息的消息队列。

3. 接口函数

中间件应用编程接口的设计原则是简洁、易用和功能全面。根据这个原则，采用分布式对象技术，首先是对分布式应用对象进行抽象，同时进行分布式应用过程的模块划分；进而设计面向消息中间件的应用编程接口，并对中间件的系统功能进行封装。中间件的接口函数有：

ConnectQM——完成运行进程到队列管理器的连接；
OpenQueue——打开一个消息队列；
SendRequest——将请求消息放入发送消息队列；
ReceiveRequest——从接收消息队列读取请求消息；
SendReply——将应答消息发送到发送消息队列；
ReceiveReply——从接收消息队列读取应答消息；
SendReport——将通知消息放入发送消息队列；
ReceiveReport——从发送消息队列中读取通知消息；
PropertySet——设置一个消息队列对象的属性；
PropertyQuery——查询一个消息队列对象的属性；
CloseQueue——关闭一个打开的队列；
DisconnectQM——将运行进程与队列管理器断开。

消息中间件的任务除了高效、可靠的传递消息之外，还应能完成安全加密解密、不同系统之间的转换、支持消息驱动处理模式的触发机制、向多个应用广播数据、发布/订阅、广泛的错误查询和错误恢复、网络资源定位、消息和请求的优先排序等。

消息中间件的功能如下：

① 无连接消息传递。使用存储和转发消息队列，计算机不会受到网络的干扰，也不必建立会话。因为在应用程序级使用无会话模式，所以源计算机和目标计算机不需支持相同的网络协议。它既可支持网际协议（IP），也可支持 Internet 数据包交换（IPX）协议。

② 消息优先化。消息优先化允许先发送紧急或重要的消息，再发送次重要消息，这样可以保证对关键的应用程序有充足的响应时间，而忽略不太重要的应用程序。

③ 有保障的消息传递。消息可以存储在基于磁盘的队列中，然后转发，以提供有保障的传递。

④ 事务处理消息。使用事务处理功能，可以将一些相关活动在一个事务处理中连接起来，保证消息按顺序传递，保证消息只传递一次并确认消息能从目标队列成功返回。

⑤ 动态队列创建。可以随意创建或更改队列属性，而不影响消息应用程序。

⑥ 消息路由。消息队列能根据网络的物理拓扑、会话集中性需要以及传输连通性提供消息路由。会话集中性易于有效地使用慢速通信链接。

⑦ 不同硬件系统的集成。消息队列可跨越各种硬件平台使用。

⑧ 跨软件平台的支持。例如 MSMQ－MQSeries Bridge 具有扩展消息队列的功能，提供与 IBM MQSeries Version2 和 Version5 平台的无缝连通性。消息队列连接器（MQC）可用于将消息队列的功能扩展到其他非 Windows 操作系统，包括 Unix、AS/400、Tandem、DigitalVMS、CICS/MVS 和 Unisys Clear Path HMP 等。

由此可见，面向消息的中间件可以为开发者提供：在不可靠的网络上实现可靠的通信；实现来自不同平台和网络协议的应用间的无缝连接；简化开发模型；直接调用发送/接收的应用程序接口，实现应用程序间的互操作，不必考虑复杂的网络通信问题，避免了系统底层的工作；克服基于 RPC 的中间件的限制，提供基于消息的异步通信机制；同时 MOM 不会占用大量的网络带宽，可以跟踪事务，通过将事务存储在磁盘上，来恢复系统及网络故障。常见的 MOM 产品有 DEC 的 MessageQ、IBM 的 MQSeries、微软的 MSMQ。

4. 消息中间件的实现与微软的 MSMQ 使用示例

首先介绍利用队列结合 WinSock 完成对消息中间件的实现。在一个过程中多次使用 Socket 控件的 SendData 方法时，假设在主窗体上有一个 WinSock 控件，名称为 Sock1，执行下面的程序段：

```
Private Sub Command1_Click ()
    Sock1.SendData "Start"
    Sock1.SendData "Run"
    Sock1.SendData "End"
EndSub
```

假如在另一端想使用 Socket 控件的 DataArrival 事件分 3 次把这 3 条信息接收到 3 个变量中，那么执行的结果和预期结果将是不一致的，执行结果只能接收到“End”。因为 Socket 控件的 SendData 方法，并不是一遇到执行就马上把消息发送过去。这时 Command1_Click 事件的进程在占用系统，在此时执行 SendData 方法，它只是把要发送的消息送入内存，并不真正开辟进程进行发送。所以第一次执行 Sock1.SendData “Start” 只是把消息 “Start” 放入内存，接着执行 Sock1.SendData “Run” 同样是把 “Run” 放入内存，但在这时放 “Run” 时，就会覆盖掉刚才放入的尚未发送的 “Start”。同理，第三次的执行又用 “End” 覆盖掉 “Run”。当执行完此过程，Command1-Click 事件的进程结束，放弃对系统的控制权时，也没有优先级更高的进程等待请求系统，这时，系统才真正执行发送消息，从内存中取出消息并发送出去。这样，在另一端就只能接收到最后一个消息。

但是在软件开发中，许多时候，又不得不在同一过程中发送多个消息，这就使得这一问题必须得到解决。在此使用队列的方法来解决这个问题。

首先定义一个定长队列，长度为此应用程序在所有发送消息的过程中连续发送的最大数目。此队列的结构见图 3-10，其中，busyFlag 表示此成员是否空闲，message 表示要发送的消息，other 表示别的一些控制，如消息的分类等，N 表示此成员的下一个成员。

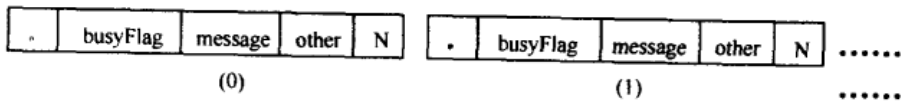


图 3-10 队列结构

其次，给此队列提供一些方法，如判空方法、入队方法、删除队头的方法及判断队列是否满的方法等等。

入队方法是先从此队列中找出一个空成员，然后判断此队列是否为空，如果为空，则队列的队头、队尾均指向此成员；否则，队尾的 N 指向此成员，新队尾指向此成员。最后将此成员的 busyFlag 置为忙，message 和 other 分别置相应的值，N 值置为 -1。这样一个成员入队完毕。

删除队头的方法是先将队头的 busyFlag 置为闲，然后将队头置为原队头的 N 值，再判断队头是否为指向空，若队头指向空，则队尾也指向空，表明此时队列为空。最后再写一个发送消息的函数，此函数的功能是把发送消息的语句 Sock1.SendData 中的“消息”不立即放入内存，而是先入队列，然后再判断此队列中是否只有一个成员，如果只有一个成员，则放入内存，否则什么也不做。

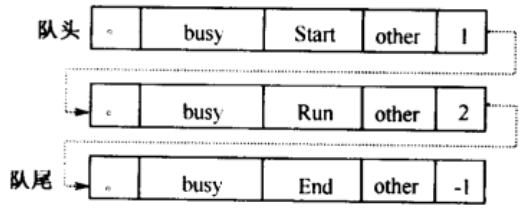


图 3-11　队列变形

这样，若执行前面的程序段，则队列变形见图 3-11。并且 Start 已经放入内存，等到程序段执行完，释放了系统的控制权后，就开始发送 Start，发送完后，把此成员从队头删除。然后判断队列是否为空，若为空，则结束发送；否则，再取队头继续发送，这时就取出 Run 发送，重复上述过程。这样，在另一端，就可以分 3 次接收到 3 次发送的 3 个消息。这样，接收到的消息就和所期望的是一致的。

因为网络的拥塞是由于消息发送的无序造成的，许多被发送的消息都挤在内存等待发送，没有一个有效的管理机制来控制它们的发送次序，就造成了网络的拥塞。使用队列的方法就是先让所有要发送的消息按发送的先后次序进入队列排队，然后从队头开始，有次序进行先来先发送，后来后发送。这样消息的发送始终是在井然有序地进行，从而成功地解决了网络拥塞。

下面对微软公司的 MSMQ 程序开发构件及应用作简要介绍。

微软的 MSMQ 是一种常用的消息中间件，主要包括如下的构件：

MSMQQuery——对现在的信息队列查询 MQIS（MSMQ Information Store），查询结果存在 MSMQQueueInfos 对象中，返回 MSMQQuery；

MSMQQueueInfos——在 MSMQQuery 返回的队列集合中选择一个公共队列；

MSMQQueueInfo——用作查询管理，创建和删除队列及改变队列属性；

MSMQQueue——用来浏览消息队列内容；

MSMQEvent——用来执行排队，等候队列活动的事件；

MSMQMessages——用来给消息队列创建 MSMQ 消息。

下面的 ASP 脚本示例显示了利用 MSMQ 实现一个简单的网上订票处理系统。发送消息的 ASP 页面将乘客的座位要求发送到队列。接收消息的 ASP 页面读队列中的消息并显示。

发送消息的 ASP 脚本：

```
<%
On Error Resume Next
Dim objMSMQ, objMsgQueue, objMessage, strCaption, strBody
'txtLabel 和 txtBody 是请求页面的文本框
strLabel = Request. Form ("txtLabel")
strBody = Request. Form ("txtBody")
Set objMSMQ = Server. CreateObject ("MSMQ. MSMQQueueInfo")
objMSMQ. PathName = ". \ Reservation Request"
objMSMQ. Label = "Reservation Request"
objMSMQ. Create
On Error Goto 0
Set objMsgQueue = objMSMQ. Open (2, 0)
If objMsgQueue. IsOpen Then
    Set objMessage = Server. CreateObject ("MSMQ. MSMQMessage")
    objMessage. Body = strBody
    objMessage. Label = strLabel
    objMessage. Send objMsgQueue
End If
objMsgQueue. Close
%>
```

读消息并显示的 ASP 脚本：

```
<%
Set objMSMQ = Server. CreateObject ("MSMQ. MSMQQueueInfo")
objMSMQ. PathName = ReservationRequest
Set objMessageQueue = objMSMQ. Open (1, 0)
Do While True
    Set objMessage = objMessageQueue. Receive (false, true, 1000)
    If objMessage Is Nothing Then ExitDo
    Response. Write ("Customer:"& objMessage. Body & "<BR>")
    Response. Write ("Seat:"& objMessage. Label & "<BR>")
Loop
%>
```

3.5.3　数据库访问中间件

1. 在 Internet 上实现数据库访问的方式

目前，实现 Internet 上数据库的访问有以下 4 种方式。

(1) 通用网关接口 CGI (Common Gateway Interface)

通用网关接口是一组关于如何在客户端 Web 浏览器、Web 服务器与 CGI 应用程序之间传递信息的规范，是 HTTP (Hyper Text Transfer Protocol) 服务器与程序进行“交谈”的一种方式。在许多应用中，由于超文本标识语言 HTML (HyperText Markup Language) 不能完全满足用户的需要，因而需要建立应用服务器。而 Web 浏览器不能直接与应用服务器程序通信。因此，需要有媒介连接 Web 浏览器、Web 服务器和应用服务器，这种媒介中最常见的有 CGI。它接受用户的输入，将其解析为应用程序能使用的变量参数；使得 Web 服务器能在应用服务器上运行；同时，解释应用服务器产生结果，并将结果送回到客户机浏览器。由此可见，应用服务器通过 CGI 与 Web 服务器相连。

基于 CGI 的 Internet 应用程序是基于 HTML 的扩展，需要在后台运行应用服务器。基于 CGI 的 Internet 应用系统通过 CGI 脚本，将应用服务器和 Web 服务器连接。

CGI 是 Web 服务器调用外部程序的接口，当用户发送一个请求到 Web 服务器，Web 服务器通过 CGI 把该请求转发给后端运行的应用服务程序，由应用服务程序将生成的结果交给 Web 服务器，Web 服务器再把结果传递到用户端显示。这种方法的缺点是，对于每一个客户机的请求，都要重新启动一个新的服务进程，其效率受到一定的影响，这可以通过专用 API 来改进。

(2) 专用 API (Application Programming Interface)

针对 CGI 程序的上述不足之处，各大 Web 服务器厂商和数据库厂商纷纷推出各自的专用 API。在 Web 服务器与数据库服务器的连接方案中，Netscape 和微软作为 Web 服务器厂商分别推出了适用于各自 Web 服务器的 NSAPI (Netscape Server Application Programming Interface) 和 ISAPI (Internet Server Application Programming Interface)。

专用 API 模式可以实现 CGI 的全部功能，并对之进行了扩展，其工作机理与 CGI 大致相同，都是通过交互式网页取得客户输入信息，然后交服务器后台处理。由于专用的 API 采用动态链接库 (DLL) 的形式，而动态链接库可以和服务器装于同一地址空间中，因此执行效率比 CGI 高。

对于 ISAPI 来说，提供了两种简单有效的方法来扩展 Web 服务器，一种被

称作 ISAPI 服务器扩展，另一种称为 ISAPI 过滤器。ISAPI 服务器扩展是一个动态链接库，可以被 Web 服务器作为一个客户程序调用和加载。ISAPI 过滤器类似与 Internet 信息服务器 IIS（Internet Information Server），它是运行在 ISAPI 使能 HTTP 服务器上的动态链接库，用于过滤传给服务器或从服务器传出的数据。具体地说，ISAPI 过滤器辅助处理 HTTP 请求过程中的各种事件，注册事件通知消息，当选择的事件发生时，过滤器被调用并可以监视和更改数据。利用 ISAPI 过滤器可以向 Web 服务器增加各种不同的功能，如验证、压缩或加密等。

当然，采用这种类型的 API 也存在一些缺陷，例如，NSAPI 与 ISAPI 相互之间不兼容，它们只能在特定的服务器和操作系统上运行；由于采用了动态链接库的形式，一旦代码质量较差就比较容易造成服务器系统的崩溃，并且进行程序设计时会更复杂等。

三大数据库厂商 Oracle，Sybase 和 Informix 提供了 Web 服务器与数据库服务器的专用接口，且都与各自的数据库产品紧密集成，因而在效率、性能以及安全性等方面都达到了较为理想的水平。但也正是由于这种紧密集成，导致了它们只能局限于各自的数据库产品之上，所以兼容性和可移植性差。

(3) JDBC 与 ODBC

为了克服专用 API 可移植性差的缺陷，SUN 公司提出了 Java 数据库互连 JDBC（Java Database Connectivity）接口方案和微软公司提出了开放数据库互连 ODBC（Open Database Connectivity）接口方案。

JDBC 实质上是一种通过 Java 语言访问结构化数据库的低层次的 API，对于一些用结构化查询语言的关系数据库尤为有效。开发人员可以在 JDBC 的基础上，设计出用户更容易理解、便于使用的更高层次的接口和工具。JDBC 技术不但提供了标准的应用程序接口来连接关系数据库这样的数据源，而且还为数据库的生产厂家提供了标准的结构，使得这些数据库生产厂家按照这些标准结构设计自己的数据库产品驱动程序。有了这些专用驱动程序的支持，这些数据库产品的用户可以方便地直接使用自己的 Java 应用程序与这些数据库产品相互通信。JDBC 也是独立于平台和数据库的。Java 语言的中性结构和 JDBC 的开放性使得程序员只需写一次程序就能让它到处运行。

开放数据库互连 ODBC 是微软公司提供的一种通用的数据库接口。ODBC 是用来执行 SQL（Structured Query Language）语句的应用程序接口，它由一组开放数据库互连接口和规范组成，易于向任何关系数据库发送 SQL 语句，从而支持对多种数据库的访问；C++、Visual Basic 等都支持开放数据库互连接口。运用开放数据库互连接口，只需要写出一个程序就能够给相应的数据库发送数据库请求。

JDBC 和 ODBC 作为一种通用的数据库接口，具有访问不同类型的数据库的

能力；同时也因为其通用性而大大地限制了其数据库访问的效率。而且使用 JDBC 和 ODBC，必须在运行客户程序的机器上安装一个相应的驱动程序，而且还要配置 JDBC 和 ODBC。这样极大地增加了分布式的客户/服务器应用的复杂性并影响其可维护性。

（4）数据库引擎

数据库引擎也是一种数据库中间件技术。与 ODBC 类似，数据库引擎也是用来执行 SQL 语句的应用程序接口，由一组数据库引擎接口组成，易于向任何关系数据库发送 SQL 语句，从而支持对多种数据库的访问。与 ODBC 不同的是，数据库引擎支持与多种数据库的直接连接，因而效率问题得到解决。但还是必须在运行客户程序的机器上安装一个数据库引擎的驱动程序，而且还要配置 BDE。这样极大地增加了分布式的客户机/服务器应用的复杂性和影响其可维护性。

2．数据库访问中的 3 层结构和数据库访问中间件概念的提出

为了解决对数据库访问中出现的问题，人们提出了 3 层应用软件体系结构。在这种软件体系结构中，整个系统由 3 部分组成：浏览器、应用服务器和数据库服务器。浏览器通过下载较小的 Applet 生成用户界面，服务器采用相对独立的中间件搭建，这些中间件完成数据传输、安全验证、并发控制等功能。只要权限允许，数据库中间件可以调用本地 JDBC 来实现对网络中任意主机上任意类型的数据库访问，这样一来就不存在下载驱动的问题。图 3-12 是只有数据库访问的 3 层结构的示意图。

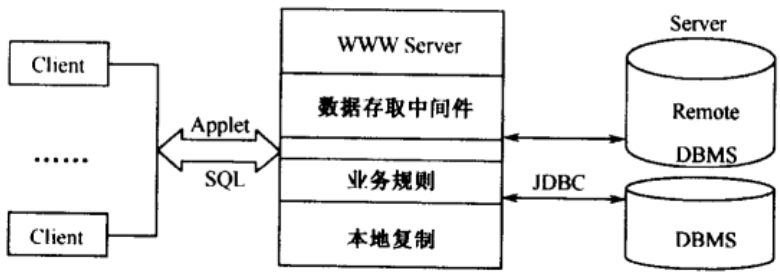


图 3-12　3 层访问数据库结构

中间层设计成多个客户机请求和管理一个或几个数据库服务器的连接，数据库访问中间件（简称数据库中间件）的主要功能包括：

① 同时管理多个客户机连接的多线程；

② 可以接收不同的厂家中立协议（从 HTTP 到 TCP/IP）客户机的连接，然后调动请求到相应具体厂家数据库服务器，并将答复返回相应客户机；

③ 可以用一组管理数据操作的业务规则进行编程，业务规则包括从限制某

部分数据的访问到确保数据正确格式再插入或更新；

④ 集中处理密集任务和将数据表达抽象到最高层，防止客户机变得臃肿；

⑤ 分开客户机应用程序与数据库系统，使公司可以自由切换数据库系统而不必重建业务规则；

⑥ 可以异步提供当前数据表或行的状态给客户机。

但这种3层结构仍存在如下几个方面的缺陷。

(1) 对连接没有充分利用

按照通常的方式，在访问局域网中DBMS时需要先建立用户连接，然后通过用户连接对数据库进行操作，操作执行完后立即断开连接。这样做虽然可以提高DBMS的对多用户的服务能力，但是建立与数据库的连接通常是相当费时的操作，这样会降低数据库服务器的响应速度，在实际使用时效率较低。

按照通常的方式，在访问DBMS时需要先建立用户连接，然后通过用户连接对数据库进行操作。但是，一般的DBMS服务器都定义了一个最大的用户数，通过这个用户数限制了同时连接到数据库服务器上的用户数量。当与数据库连接的用户数量达到这个上限时，新的用户连接请求就会失败。在Intranet应用中，网络上会时常出现大量用户同时访问同一个数据库服务器的情况，并且每个用户在一段时间内会保持这种服务连接，从而限制了可用的连接数。但是，当用户在进行客户端的处理时，与数据库的连接通常是空闲的，如果能够充分利用这段空闲时间，使得这些连接能够为其他用户提供服务，那么，就能大大提高同时服务的用户数。如果在用户不使用这些数据库连接时就断开数据库连接，虽然可以提高数据库服务器的多用户服务能力，但是建立数据库连接通常是相当费时的操作，因为它要建立某种通信连接，启动服务进程，并进行权限检查等，这样反而会降低数据库服务器的服务响应速度。可以采取两种方法来解决以上问题，一是采用连接池（在中间件中设置连接管理模块，预先在DBMS与连接管理模块之间建立一定数量的数据库连接，每个连接可以一个线程实现，组成连接池）；二是使用独立的数据库连接管理进程提供连接管理的服务，客户不是直接与数据库服务器建立连接，而是通过数据库连接管理进程建立连接，各种数据库操作也是通过该数据库连接管理进程完成的。实际上就是在客户应用端与数据库服务器端之间再增加一层中间层来负责完成连接管理任务，这就是所谓的数据库访问中间件。

(2) 对重复的数据库操作没有优化

在大型应用中，常常有一些执行频率较高的数据库操作（如查询），目前的数据库系统，对多个用户的同一种操作仍采用单独连接的方式，这种重复连接不仅更容易造成网络拥塞，而且也加重了数据库服务器的负担。这可以采用在数据库中间件中开辟缓冲区（用来对使用频率较高的数据库操作结果进行缓存）的方

法来解决。

(3) 远程访问不可靠

在广域网环境下，通过 JDBC 直接与远程数据库连接从技术上来说是可行的。但在整个操作过程中需要一直保持与远程数据库的连接，直到操作结束，并将结果返回本地为止。这么做，一方面由于带宽和流量的不同，连接的可靠性很难预测，一旦连接中断，就意味着操作的失败或数据丢失；另一方面在远程数据库服务器执行操作的过程中，并不涉及网上数据传输，而这时连接依然占用着信道，造成了连接的无谓浪费，也容易造成网络阻塞。

针对以上缺陷，可利用 RMI 将原来的 JDBC 直接连接数据库转变为两个中间件之间的连接的方式来解决。这样一来，只有调有远程对象或接受结果时才需要占用信道，这种占用是临时性的，不需要维持长时间的连接，从而减轻了网络负担，提高了访问效率。

3. 数据库中间件

由上可知，数据库中间件专门负责和完成对数据库访问操作的优化、用户连接数的管理、保证访问的安全性和可靠性，以及实现应用对来自不同厂商的数据库的访问等，进而提高对数据库访问的效率。

简而言之，数据库中间件就是指一切连接应用程序和数据库的软件。与一般的中间件一样，面向数据库的中间件允许开发人员通过单一的、定义良好的 API 访问另一台计算机上的数据库资源。用数据库中间件完成对数据库的访问有直接访问和数据库复制两种形式。

直接访问，就是应用程序像访问本地数据库一样可以直接访问和更新位于远端的数据库。中间件提供了 SQL 语义风格的语言访问，能够支持 SQL 语言的关系数据库，这样就可以像访问本地数据库一样实现对远程数据库的修改和查询。数据库中间件能实现对传统文件形式数据库的访问，这是通过支持 SQL 的数据库与文件形式的数据库之间的映射来实现的。数据库中间件还允许应用登记数据库记录的变化，从而从一个远端工作站上显示所关心的数据的变化。直接访问形式上虽然比较简单，但也有一定的局限性，如应付大量用户的直接请求对硬件提出了很高的要求，网络拥挤会延长响应时间。数据库复制中间件则解决了这一问题。

在数据库复制中，主要采用缓存技术，可分为两种情况，客户端的缓存和服务器端的缓存。虽然这两种技术的目的都是为了重用数据，但服务器上缓存的实现与客户端的缓存是有区别的。在缓存读取上，客户端上的缓存在读数据时可以通过比较版本来判断缓存数据的有效性；而服务器上，可以考虑“双向”的缓存，即保证服务器上缓存的内容一直与数据库保持一致，于是，服务器缓存成为

客户端读取数据库数据的惟一通道。所谓“双向”，指客户端既可以直接往服务器缓存里写数据，也可以直接从服务器缓存中读数据，不像客户端缓存那样在缓存数据无效时，需要转向远程服务器索取数据，然后再更新缓存。要实现“镜像”式缓存数据的高度一致性，可以采用主动更新的方法，即不管客户端是否有服务请求，服务器根据缓存的内容定时从数据库读取最新数据，保证一般情况下缓存的内容和数据库都是一致的。这样的方法不能用到客户端缓存，因为客户端数目众多而且情况复杂，即使定时从数据库读取数据也会由于网络传输延时等原因不能确保数据的及时更新。在数据库复制中刷新滞后是一个很重要的问题。

4. 数据库中间件的类型

目前面向数据库的中间件有好几种类型，但基本上可归为三大类，即本地中间件、呼叫层接口 CLI（Call Layer Interface）和数据库网关。

由于本地中间件是为特定的数据库设计的，所以能够提供最佳的访问性能。如 Sybase 设计的从C++访问 Sybase 数据库的中间件就是一个面向数据库的高性能的本地中间件。但是，应用程序一旦用本地中间件建立了数据库连接，当要改变数据库时，需要对应用程序进行很大的修改。

呼叫层接口为多个数据库提供了统一的界面，如 ODBC 和 JDBC 等都是基于 X/开放 SQL 的呼叫层接口。它可以把一般通用的接口呼叫转换成任意的数据库本地语言，也可以将响应集再转换成一致的表现形式，以便发出请求的应用程序理解。

数据库网关的主要功能是完成不同数据库模型的转换，能提供对大型系统内部数据的访问。数据库网关可以从统一的应用程序接口集成多个数据库，以便访问和映射旧的数据库模型，并且对出入数据库网关的查询和信息进行转换。

5. 几种常见的数据库中间件描述

(1) ODBC

数据库管理系统发展到今天，可以说已经达到了极致，诸如国际国内的主流数据库管理系统 Oracle、Sybase、Informix、Ingres、DB2 等，数据库系统的技术已经非常成熟，不同的数据库管理系统都占据着各自的市场。这给人们带来更大的选择自由度，同时也带来了不少问题。

对于管理信息系统和数据库应用系统的开发，人们往往根据实际需求和习惯喜好采用不同的数据库系统。应用这些系统，各单位各部门投入了大量的人力、物力，相继完成了一些管理系统。为了保护过去的投资，利用已建立的信息系统，迫切需要一种能访问多种数据库的操作平台，以便建立更大、更完备、更全面的信息管理系统。

为了适应这种需求，微软推出了开放数据库互连技术 ODBC。开放数据库互连技术实际上是一个数据库访问库，它包含访问不同数据库所要求的 ODBC 驱动程序。应用程序要操作不同类型的数据库，只要调用 ODBC 所支持的函数，动态链接到不同的驱动程序上即可。随着 ODBC 技术的推出，许多开发工具软件都把 ODBC 技术集成到自己的软件中，如 Visual Basic、Visual C++、PowerBuilder 等。

开放数据库互连 ODBC 是微软公司开放服务结构 WOSA（Windows Open Services Architecture）中有关数据库的一个组成部分，它是一个 CLI，通过允许开发者制作一个在大多数关系数据库中可运行的简单的 API 调用，简化了从 Windows（以及其他一些操作系统）到数据库的访问。ODBC 建立了一组规范，并提供了一组对数据库访问的标准 API，这些 API 利用 SQL 来完成其大部分任务。ODBC 本身也提供了对 SQL 语言的支持，用户可以直接将 SQL 语句送给 ODBC。由此可见，ODBC 实际上并不是一个产品，而是微软在几年前创建的一个标准。

一个基于 ODBC 的应用程序对数据库的操作不依赖任何 DBMS，不直接与 DBMS 打交道，所有的数据库操作由对应的 DBMS 的 ODBC 驱动程序完成。也就是说，不论是 FoxPro、Access 还是 Oracle 数据库，只要有相应的 ODBC 驱动程序支持，均可用 ODBC API 进行访问。Microsoft Developer Studio 为大多数标准的数据库格式提供了 32 位 ODBC 驱动器，包括 Oracle、SQL Server、Access、Paradox、dBase、FoxPro、Excel 以及 Microsoft Text 等。由此可见，ODBC 的最大优点是能以统一的方式处理几乎所有的数据库。

从中间件的角度来看，ODBC 是基于数据库的中间件标准。通过 ODBC 访问数据库的方式是绝大多数应用程序使用数据库的方式，由于驱动程序与具体的数据库有关，它是一个用以支持 ODBC 函数调用的模块（通常是一个 DLL）。应用程序通过调用驱动程序所支持的函数来操作数据库，若想使应用程序操作不同类型的数据库，就要动态地链接到不同的驱动程序上。ODBC 具有良好的数据库独立性，它可以避免应用程序对不同类型数据库使用不同的 API，通过 ODBC 可以使得数据库的更改变得非常容易，因为对应用程序来说这只需改换一下驱动程序。ODBC 结构如图 3-13 所示。

像所有的中间件一样，ODBC 提供一个定义良好的、不依赖于数据库的 API。使用 API 时，ODBC 通过一个驱动程序管理器来判定应用程序要连接的数据库的类型，并载入（或卸载）适当的 ODBC 驱动，这样，就实现使用 ODBC 的应用程序和数据库之间的相互独立。

ODBC 目前提供 32 位版本。大多数数据库都有 ODBC 驱动。ODBC 是免费的，而其驱动程序不是免费的。这些 ODBC 驱动程序可以从数据库供应商或第

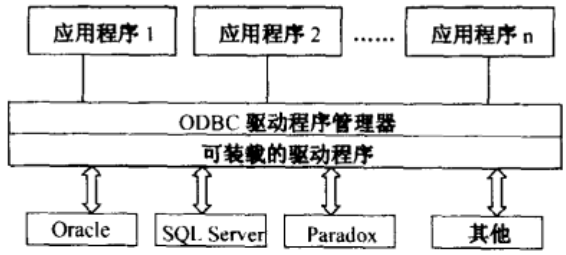


图 3-13 ODBC 结构图

三方提供商购买。流行的应用程序开发工具大多通过 ODBC 提供对数据库的访问。实际上，ODBC 是微软 Visual C++ 以及其他开发工具与数据库连接的惟一途径。

(2) OLE DB

OLE DB 常被戏称为 ODBC 的“大哥”，它定义了一个数据访问服务器的集合，通过这些服务器可以很容易地连接到任意数目的数据源。这样，开发者就可以把多种不同的数据源作为单一的虚拟数据库来管理。OLE DB 允许使用标准的 COM 接口访问数据。

OLE DB 是一种技术标准，目的是提供一种统一的数据库访问接口。OLE DB 接口能访问各种 DBMS 和非 DBMS 信息源，这些信息源包括主机数据库（如 IMS 和 DB2 等）、服务器数据库（Oracle 和 SQL Server 等）以及桌面数据库（微软的 Access 等）。非 DBMS 信息源包括存放在 Windows NT 和 Unix 文件系统中的信息、索引顺序文件、电子邮件、电子表格、Web 上的文本或图形以及目录服务等。OLE DB 使得数据库的消费者（应用程序）可以使用同样的方法访问各种数据，而不需要考虑数据的具体存放地点、格式和类型。

OLE DB 将传统的数据库系统划分为多个逻辑部件，部件之间相互独立又相互通信。其中消费者（consumers）是使用 OLE DB 接口对存储在数据提供者中的数据进行控制的应用程序，如典型的数据库应用程序以及需要访问各种信息源的开发工具或语言等。提供者（providers）是暴露 OLE DB 接口的软件部件。提供者大致分为两类，即数据提供者（data providers）和服务提供者（services providers）。数据提供者是提供数据存储的软部件，例如，关系型 DBMS、存储管理器、电子表格、ISAM 以及电子邮件等都可以是数据提供者。服务提供者位于数据提供者之上，是从 DBMS 中分离出来且能独立运行的功能部件，如查询处理器和游标引擎等，这些部件将使数据提供者提供的数据能以表格状的形式向外表示，并实现数据的查询和修改功能。服务提供者不拥有数据，但通过使用 OLD DB 接口生产和消费数据来封装某些服务。

OLE DB 的具体实现是一组C++ 的 API 函数，只能供C++ 调用。使用 OLE

DB的API，可以编写出访问符合OLE DB标准的任何数据源的应用程序，还可以编写出针对某些特定数据存储的查询处理器和游标引擎。例如，SQL Server 7.0的查询处理器是使用OLE DB的API编写的。

(3) JDBC

SUN公司的JDBC是第一个支持Java语言的数据库应用程序接口，功能上与ODBC相仿，提供Java开发人员一个从支持Java开发或支持Java应用程序运行的环境访问各种数据库的统一的接口。JDBC API使开发者不必不断重写程序而可以建立数据库前台。

JDBC向应用程序提供了独立于数据库的统一的API接口，其奥秘是一组由驱动程序实现的Java接口。驱动程序负责标准JDBC调用想支持的数据库所要的具体调用转换。也就是说，JDBC主要由两层组成：JDBC应用程序接口（JDBC API）和JDBC驱动应用程序接口。JDBC API提供从应用程序到JDBC管理器的通信。开发者通过API使用标准Java机制访问数据库。数据库供应商提供JDBC驱动接口，也可以通过JDBC-ODBC连接桥使用传统的ODBC连接。基于JDBC实现的应用程序可以不变，驱动程序则各不相同，驱动程序可以用来开发多层数据库设计中的中间层，也就是数据库中间件。

JDBC API定义了一个Java类集，允许Applet、Servlet、Java Bean和Java应用程序连接数据库。通常，由这样一个小应用通过网络连接远程关系数据库服务器，如Syabas、Oracle或Informix。这些由数据库供应商提供的原始的Java JDBC类与用户自定义的应用程序类共存，提供一种“纯Java”的、可移植的数据库访问。这样就允许从任意支持Java的平台到任意数据库的连接。

除了向开发者提供统一的独立于DBMS的框架外，JDBC还提供了让开发者保持数据库厂家提供的特定功能的办法。也就是说，JDBC允许开发者直接将查询字符串传到连接的驱动程序，这些字符串可能是ANSI SQL，也可能不是。对这些字符串的使用由基础驱动程序负责。

JDBC管理器和ODBC管理器一样，按Java小应用或应用程序的需要载入或卸载数据库驱动器。JDBC支持单个或多个数据库服务器的连接。就是说，一个小应用可以同时连接本地的所有数据库和Internet上的公用数据库。

JDBC API遵循标准Java安全机制。简而言之，应用程序被看成是信任代码，小程序被看作是不信任的代码。一般地说，编写安全JDBC驱动程序的工作留给驱动程序厂家。Java虚拟机对不信任小程序采用自己的安全检查并进行建档。但如果JDBC驱动程序厂家要增加驱动程序特性以扩展模型，例如多个小程序用同一个TCP接插连接与数据库对话，则厂家要负责检查每个小程序是否允许使用这个连接。

当然，数据库中间件的引入，可以全面完成以上功能，进而杜绝和防止数据

库在安全方面的隐患。

(4) 数据库引擎

Borland 公司的数据库引擎也是一种数据库中间件技术，可参见本节前面部分。

(5) 数据库网关

按照通常的方式，在开发分布式的客户机/服务器应用时，涉及到远程数据库访问时往往采用开放数据库互连方法或数据库引擎的方法。在使用 Visual Basic 或 C++ 开发应用程序时，ODBC 是首选的方法；在使用 Delphi 开发时，DBE 是最合适的选择。但是无论是使用开放数据库互连还是使用数据库引擎，在应用运行的主机上必须安装一个开放数据库互连接口（通常是 ODBC）或安装一个数据库引擎（通常是 BDE）；此外，还需要配置相应的数据库接口或数据库引擎。在一个本地集中的应用环境中，这种方案是可行的。然而，在一个分布的应用环境中，尤其是跨地域的应用环境中，这种方案的可行性值得怀疑：系统的复杂性急剧增加，系统的可维护性大大降低。

如果在分布的应用环境中，在访问数据库时也无需在本机上配置数据库访问接口，如上所述的制约系统因素将不复存在，从而大大提高系统的可维护性和降低系统的复杂性。设计数据库网关的目的就在于此：在客户机上实现数据库访问的零配置方案，把数据库接口的复杂性集中在数据库服务器上；同时加入适当的安全控制策略，增强系统的安全性。

数据库网关（也叫 SQL 网关）是一种应用程序接口，开发者只需使用一个 API 调用，数据库网关就可以完成所有的其他工作。数据库网关通过使用同一接口完成对运行在多种平台上的不同数据库的访问，提供给开发者到任意数目数据库的访问接口，包括一些运行在典型的不易访问的环境下的数据库。如通过一个数据库网关，可以同时访问存储在大型主机环境下的 DB2 数据库、小型机上的 Oracle 数据库和 Unix 服务器上的 Sybase 数据库。在实际使用中，数据库网关把 SQL 调用解释成为标准 FAP 格式。FAP 格式是通用的客户机和服务器连接，也是异质数据库和运行平台的通用连接。网关可以把 API 调用直接翻译成 FAP，把请求传递到目标数据库并翻译以便目标数据库和平台作出响应。

目前市场上有很多数据库网关产品，如 EDA/SQL、RDA、DRDA 等。

RDA 并不是一种产品，它是一个开发者访问数据的标准。RDA 使用 OSI 并且支持动态 SQL 语句，允许客户端同时连接一个以上的数据库。但它不支持典型的事务相关服务。

DRDA 是 IBM 的一个数据库连接标准，它有许多数据库的支持，如 Sybase，Oracle，IBI 和 Informix。与其他的数据库网关一样，DRDA 提供便利的运行在多平台环境下的任意数目数据库的连接。DRDA 把数据库任务定义为远程请求、远

程工作单元、分布式工作单元和分布式请求。DRDA 是一个定义良好的标准，它要求数据库符合标准的 SQL 语法，以便能够充分发挥 DRDA 在不同的系统、不同的情况下运行不同的数据库的功能。

在基于数据库中间件领域中，目前还提出了应用分割技术，即将用户的一些应用逻辑放到中间层，为客户机“减肥”，这也为 NC（Network Computer）等的引入打下了基础，并增强了应用程序的处理性能、安全性和并发性。目前，很多数据库前端开发工具都支持应用分割技术。

但是，在基于数据库的中间件模型中，数据库作为信息的中心存储单元，中间件负责数据间的同步及点到点通信。这种方式不适合于高性能应用处理，因为它需要大量的数据通信，同时，当网络发生故障时，系统将不能正常工作。

6. 数据库中间件的优、缺点

数据库中间件具有中间件普遍的优点，如下所述：

① 移植性好。中间件封装了各种与平台有关的细节，使更换操作系统和通信协议等底层的配置无需改变应用程序代码。

② 集成方便。中间件可以非常容易地集成到应用开发环境中，无需大的代码改动。

③ 易于扩充。中间件的局部改进和整体升级只要保持对外接口不变就不会影响到系统的其他部分，在功能上对应用程序实现了透明性。

④ 使用简单。中间件对各种数据源使用统一的访问方法，使用户不必关心数据库选择等繁琐的操作，降低了用户参与程度。

总之，利用数据库中间件可以实现客户端的零配置方案；降低客户端的数据库接口的复杂性；增加了用户访问数量；支持与多种关系数据库的连接；增强分布式客户机/服务器应用程序的可维护性；加强了数据库访问的安全机制。

但数据库中间件的使用，也带来一些缺点：数据库操作比较集中，统一由数据库中间件负责数据间的同步及点到点通信，对数据库中间件的可靠性要求就非常高，一旦中间件出现问题，所有数据连接都将断掉，从而导致系统瘫痪。这种方式尤其不适合于高性能应用处理，因为它需要大量的数据通信，对数据库中间件的性能要求将更加严格。

7. 一种数据库中间件原型的构造方法

图 3-14 为一种数据库中间件的构造原型方案。它主要由 5 部分组成：客户交互模块、远程访问模块、安全控制模块、缓冲区、操作评价模块和连接管理模块。这里的模块是一个广义的范畴，可以是一段程序，也可以是一个构件。

该原型的工作机理是：客户交互模块完成接受客户请求，寻找数据源和结果

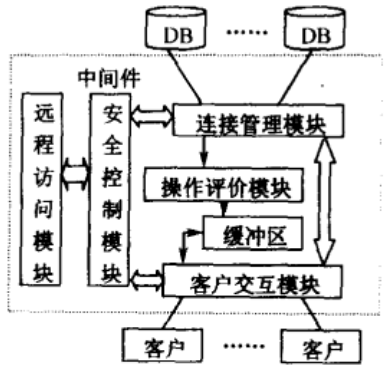


图 3-14 数据库中间件结构

回送的功能。当客户发来 SQL 请求时，它先判断数据源是本局域网数据库还是远程数据库（即远程数据库）。如需访问远程数据库，则将用户请求送远程访问模块，该请求被送到远程主机，作为一个客户端请求给那里中间件的客户交互模块。数据源是本地数据库的，如果缓冲区中有与该请求要进行的操作相同的操作，可以直接与缓冲区交互取得结果。否则就需要送连接管理模块处理。对于需要返回结果的操作，客户交互模块还负责将结果回送给客户，这里需要注意的是对于远程客户，需要将结果回送给远程中间件的远程访问模块处理。

其中，操作评价模块根据操作评价算法对一段时间内的数据库操作使用次数评估，视缓冲区大小，取前几名操作及其结果送缓冲区。并负责一定时间间隔后重新评估，刷新缓冲；连接管理模块负责连接池的创建与删除，连接的分配与冲突仲裁，请求排队等工作，并能记录一段时间内的数据库操作，作为操作评价模块的信息来源；安全控制模块负责对数据库访问中的出入信息的过滤。

8. 通用数据库中间件的概念模型

目前使用 CORBA 来实现数据库访问技术，较为流行的做法是将 CORBA 对象作为数据库的前端，根据数据库结构构造相应的 CORBA 对象，使用 IDL 语言描述数据结构，客户端利用 IDL Stub 与 CORBA 对象通信，而由 CORBA 对象实现数据库中数据的存取和查询，这是一种较为直观的解决途径，但是它缺乏通用性，它需要针对数据库的结构和实体的内容来编写相应的对象。

例如，在“数字城市”软件构件的开发过程中，整个领域构件开发都是围绕领域数据进行的。实际上即使同一领域数据在形式也是有一定差别的，要想开发出具有高复用性的大粒度构件是比较困难的。如果仅仅是将数据进行对象包装处理显然是难以满足要求的。因此以领域数据为中心的数据存取中间件是解决这些问题的重要途径。

特定领域内知识具有相对一致性的特点是建立一个领域内通用标准的前提。图 3-15 示出了针对领域数据的通用数据中间件，通过在领域软件构件和数据源之间建立一个中间层，对构件屏蔽数据源的差异。中间件向构件提供一致的数据源视图，完成从实际数据源到数据源视图的转换。

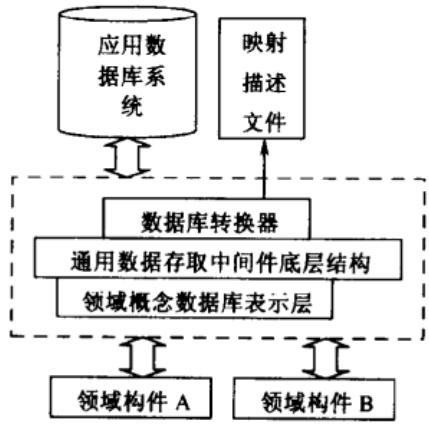


图 3-15　通用数据库存取中间件的概念

通用数据存取中间件底层：负责数据库转换器与领域概念数据库之间的通信、内部功能调用。

领域概念数据库的表示层：主要是将数据视图及提供的服务以一定形式提供给领域构件。

映射描述文件：描述领域概念数据库中定义的数据视图与应用数据库中的数据结构之间的映射关系，使概念数据知道它所描述的特征在具体应用数据库中存放的形式、位置。

数据库转换器：数据库转换器读取相应的映射描述文件，根据一定的规则，将应用数据库系统中的数据转换后经由领域概念数据库表示层传送给领域构件；同时，领域构件对数据的任何修改都要通过领域概念数据库表示层，经由数据库转换器反映到相应的应用系统数据库中。

3.5.4　对象中间件

对象中间件也称基于对象构件模型的中间件或称对象请求代理中间件 ORBM（Object Request Brokers Middleware）。对象请求代理是近年来才发展起来的一项新技术，它可以看作和编程语言无关的面向对象的 RPC 应用，被视为从面向对象过渡到分布式计算的强大推动力量。目前有两种对象请求代理的标准，分别是 CORBA 和 DCOM，这两种标准是相互竞争的，而且两者之间有很大的区别，这在一定程度上阻碍了对象请求代理中间件的标准化进程。

面向对象一直是软件界努力追求的目标，传统的对象技术通过封装、继承和多态提供了良好的代码重用功能。但是这些对象只存在于一个程序中，外面的世界并不知道它们的存在，也无法访问它们。面向对象的中间件就是解决这些问题，它提供了一个标准的构件框架，能视不同厂家的软件通过不同的地址空间、网络和操作系统互相交互访问。该构件的具体实现、位置及所依附的操作系统对客户来说都是透明的。面向对象的中间件技术的目标就是为软件用户和开发者提供一种应用级的即插即用的互操作性。

ORB 是一个在对象间建立客户/服务器联系的中件。使用 ORB，客户可以调用服务器的对象或对象中的应用，被调用的对象不要求在同一台机器上。由 ORB 负责进行通信，同时 ORB 也负责寻找适于完成这一工作的对象，并在服务器对象完成后返回结果。客户对象完全可以不关心服务器对象的位置，实现它所采用的具体技术和工作的硬件平台，甚至不必关心服务器对象的与服务无关的接口信息，这就大大简化了客户程序的工作。既然能够这么方便，那 ORB 就需要提供在不同机器间应用程序间的通信，数据转换，并提供多对象系统的无缝连接。

我们通常编制客户/服务器程序时，常常需要自己定义通信协议，而协议的制定往往与硬件和实现的方法有关，而 ORB 能够简化这一过程。在 ORB 下，协议通过 IDL 语言进行定义，保证了一致性，为了照顾到灵活性，ORB 允许程序员选择相应的操作系统，执行环境和编程语言。更重要的是它可以使原来的代码通过一定的方式重用。CORBA 是面向对象标准的第一步，有了这个标准，软件的实现与工作环境对用户和开发者不再重要，可以把精力更多地放在本地系统的实现与优化上。

ORB 的工作原理是：ORB 能实现分布环境中位于不同机器上应用之间的互操作以及多对象系统之间的无缝连接。在传统的 C/S 应用中，开发者使用自己设计的标准或通用标准来定义设备之间的协议。协议定义与实现的语言、网络传输、消息定义格式及其他因素有关。ORB 简化了这一过程，它使用 IDL 来定义应用接口之间的协议。ORB 允许程序员选择通用操作系统，运行环境和编程语言。

目前，ORB 存在两个彼此竞争的标准：CORBA－ORB 和 DCOM－ORB。当使用 ORB 时，IDL 用于定义对象之间的接口，它类似于 RPC 中的 IDL 定义过程的接口。ORB 特别适用于对象接口变化不频繁，不会导致代码经常被重新编译及链接的情况。ORB 的总体框架图如图 3-16 所示。

有关详细情况见第 4 章。

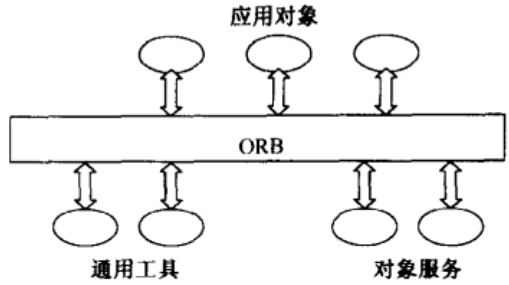


图 3-16　ORB 的总体框架结构

3.5.5　交易中间件

交易中间件也称分布式事务处理中间件 DTPM（Distributed Transaction Processing Middleware）。交易中间件是一个联机事务处理平台软件。它主要是为应用程序提供运行环境及各种服务，如程序加载、程序启动、负载均衡、出错恢复及一些应用管理功能。它是专门针对联机交易处理系统而设计的，联机交易处理系统需要处理大量的并发进程，涉及到操作系统、文件系统、编程语言、数据通信、数据库系统、系统管理和应用软件，是一个相当艰巨的任务，但是可以通过采用交易中间件来简化。交易中间件就是一组程序模块，可以大大减少开发联机交易处理系统所需的编程量。

1. 事务的概念

联机事务处理的基本概念是事务（transaction）或称交易。一个完整的事务是指一个程序或程序段，在一个或多个资源上为完成某些功能的执行过程。我们将一个完整的事务描述为 T：a_1 a_2… a_n，此处 a 是这样一些操作：读、写、删除、重写、打开、关闭、事务开始、事务提交、事务作废等。其中 a_1 是事务的开始，最后一个 a_n 必须为事务的提交。系统中的所有事务由一个事务管理系统所管理，事务开始操作时被赋予一个惟一的标识符 TID（Transaction Identify），在一个事务的生命期内此标识由事务管理系统惟一标识一个事务。

事务的提交表示一个事务的结束。一个已提交的事务如果想撤消（undo），必须通过另一个事务进行，事务的开始和事务的提交之间成为一个事务的活动状态，一个事务失败后必须通过回退（rollback）使事务回到事务开始前的状态。确切地说，事务具有以下特性（ACID）：

① 原子性（atomicity）：一个事务涉及的所有操作，要么全部成功，或者回退到原来的状态。

② 一致性（consistency）：一个事务把一个对象从一个合法的状态转到另一

个合法的状态，如果事务失败，必须把对象恢复到前一个合法的状态。

③ 隔离性（isolation）：一个事务对共享数据库所做的操作，在本事务提交之前对其他事务来说是不可见的。

④ 永久性（duribility）：一个成功的事务，除非运行另一个事务来改变它，否则，其结果是不能改变的。

2. 事务管理系统

事务处理涉及到操作系统、文件系统、编译语言、数据通信、数据库系统、应用软件的方方面面，是一个相当艰巨的任务，但是工作的任务可以通过事务管理系统来简化。可以大大减少事务管理系统应用所需的编程量。

数据通信子系统也是一个完整的联机事务处理的应用重要组成部分，尤其对分布式的应用更是这样。通信子系统和事务管理子系统一起处理需要多处提交的交易，这类交易涉及远程存取、由一台机器上的一个事务管理系统通过发送消息通知另一台机器上的别的事务管理系统。通信子系统将提供消息发送、名字服务和网络接口等。

3. 交易中间件

X/OPEN 组织专门定义了分布式交易处理的标准和参考模型，把一个联机交易系统划分为资源管理（RM）、交易管理（TM）和应用（AP）三部分。定义了应用程序、交易管理器多个资源管理器是如何协同工作的。资源管理器是指数据库和文件系统，交易管理器可归入交易中间件。交易中间件管理由应用声明和提交的交易，并通过两阶段提交协议等方式来保证分布式交易的完整性、控制并发、实现交易路由和均衡负载。

交易中间件在理论上比较成熟，功能和性能界定清晰，基本上适应于联机交易系统，如银行业务系统、定票系统等。尽管交易信息也是消息，交易中间件也是基于消息的传输，也可支持同步和异步方式，但与消息中间件的定位差距较大，属于一种专用的中间件。采用交易中间件给系统带来以下主要好处：

① 提供一个预先定义的应用框架和模型，允许开发者以板卡/插槽的开发方式进行开发，提供很强的模块化结构；

② 可以减少系统的复杂性（包括屏蔽硬件、网络环境、异构数据库等）；

③ 可以有效完成负载均衡；

④ 能保证分布式交易的完整性；

⑤ 能保证应用软件在不同平台上的平滑移植；

⑥ 能将前面的大量网络连接汇接成较少的后台连接，降低网络负担，提高数据库效率。

4. TPMonitor

TPMonitor 是一种复杂的中间件产品（即中间件的集成），它为应用处理提供了一种通信机制，它允许开发者在 TPMonitor 环境中定义事务服务。TPMonitor 位于客户机和数据库服务器之间，采用 3 层或多层模型。客户通过 TRPC（Transaction RPC）机制在 TPMonitor 中调用事务，TPMonitor 运行事务来连接数据库，并将处理结果返回给客户端。

在 TPMonitor 中，事务有一个明确的起止点，如果事务失败，TPMonitor 可以回滚事务，不会使系统处于不完整、不一致状态。TPMonitor 同时可以复用数据库请求。因为每个客户调用事务，而不是直接和数据库进行连接，因此 TPMonitor 可以协调数据库请求，传统的每个客户的连接（connection-per-client）的限制（在客户机/服务器环境中）可以去掉，如 100 个客户可能只需要 10 个数据库连接。并且 TPMonitor 还可以在同一个事务中读写异构数据库中的信息，并保持异构数据库的完整性。常见的 TP 产品有：BEA 的 Tuxedo、IBM 的 CICS、NCR 的 TopEnd 和微软的 MTS 等。

3.6　当前支持服务器端中间件的平台技术

企业分布式应用一般具有联机事务处理、用户请求并发、数据安全性等需求，在三层结构中，它们集中体现在服务器端业务逻辑层上。通过将企业分布应用中的业务逻辑与客户端表现、数据库服务器相分离，增加了系统的可伸缩性、可用性、可靠性等性能，同时服务器端业务逻辑层的应用开发也就脱离了与客户端和数据库服务器的捆绑，而变得相对独立起来。因此，服务器端业务逻辑层是 3 层结构的核心层，在应用系统中起着至关重要的作用。

中间件技术是构件技术在服务器端业务逻辑层的应用，使得基于中间件技术的分布式系统的开发者摆脱了传统的底层网络编程和复杂的分布式事务管理的困扰，从而简化了分布式处理的复杂程度。然而，中间件开发相对客户端的构件开发要难得多，除了业务逻辑的实现之外，还有大量的服务器的事务逻辑，如多线程度发、遗留应用集成、安全认证、数据库访问及对象管理等。

考察当前主流的分布计算技术平台，主要有 Microsoft DNA 2000、SUN 的 J2EE 和 OMG 的 CORBA，它们都是支持服务器端中间件技术开发的平台，但都有其各自的特点。

3.6.1　Microsoft DNA 2000

Microsoft DNA 2000（Distributed interNet Applications）是微软在推出 Win-

dows 2000 系列操作系统平台基础上，在扩展了分布计算模型，以及改造 Back Office 系列服务器端分布计算产品后发布的新的分布计算体系结构和规范。

在服务器端，DNA 2000 提供了 ASP、COM、Cluster 等的应用支持。目前，DNA 2000 在技术结构上有着巨大的优越性。一方面，由于微软是操作系统平台厂商，所以，DNA 2000 技术得到了底层操作系统平台的强大支持；另一方面，由于微软的操作系统平台应用广泛，支持该系统平台的应用开发厂商数目众多，因此在实际应用中，DNA 2000 得到了众多应用开发商的采用和支持。

DNA 2000 融合了当今最先进的分布计算理论和思想，如事务处理、可伸缩性、异步消息队列、集群等内容。DNA 使得开发可以基于 Microsoft 平台的服务器构件应用。其中，如数据库事务服务、异步通信服务和安全服务等，都由底层的分布对象系统提供。

以微软为首的 COM/DCOM/COM + 阵营，从 DDE，OLE 到 ActiveX 等，提供了中间件开发的基础，如 Visual C、Visual Basic、Delphi 等都支持 DCOM，包括 OLE DB 在内新的数据库存取技术，随着 Windows 2000 的发布，微软的 COM/DCOM/COM + 技术，在 DNA 2000 分布计算结构基础上，展现了一个全新的分布构件应用模型。首先，COM/DCOM/COM + 的构件仍然采用普通的 COM（Component Object Model）模型。COM 最初作为微软桌面系统的构件技术，主要为本地的 OLE 应用服务，但是随着微软服务器操作系统 NT 和 DCOM 的发布，COM 通过底层的远程支持使得构件技术延伸到了分布应用领域。COM/DCOM/COM + 更将其扩充为面向服务器端分布应用的业务逻辑中间件。通过 COM + 的相关服务设施，如负载均衡、内存数据库、对象池、构件管理与配置等等，COM/DCOM/COM + 将 COM、DCOM、MTS 的功能有机地统一在一起，形成了一个概念、功能强的构件应用体系结构。而且 DNA 2000 是单一厂家提供的分布对象构件模型，开发者使用的是同一厂家提供的系列开发工具，这比组合多家开发工具更有吸引力。

但是它的不足是依赖于微软的操作系统平台，因而在其他开发系统平台（如 Unix、Linux）上不能发挥作用。

3.6.2 SUN 的 J2EE

为了推动基于 Java 的服务器端应用开发，SUN 在 1999 年底推出了 Java 2 技术及相关的 J2EE 规范，J2EE 的目标是：提供平台无关的、可移植的、支持并发访问和安全的、完全基于 Java 的开发服务器端中间件的标准。

在 J2EE 中，SUN 给出了完整的基于 Java 语言开发面向企业分布应用规范，其中，在分布式互操作协议上，J2EE 同时支持 RMI 和 IIOP，而在服务器端分布式应用的构造形式上，则包括了 Java Servlet、JSP（Java Server Page）、EJB

(Enterprise Java Bean) 等多种形式，以支持不同的业务需求，而且 Java 应用程序具有“Write once，run anywhere”的特性，使得 J2EE 技术在发布计算领域得到了快速发展。

J2EE 简化了构件在服务器端应用的复杂度，虽然 DNA 2000 也一样，但最大的区别是 DNA 2000 是一个产品，J2EE 是一个规范，不同的厂家可以实现自己的符合 J2EE 规范的产品，J2EE 规范是众多厂家参与制定的，它不为 SUN 所独有，而且其支持跨平台的开发，目前许多大的分布计算平台厂商都公开支持与 J2EE 兼容技术。

EJB 是 SUN 推出的基于 Java 的服务器端构件规范 J2EE 的一部分，自从 J2EE 推出之后，得到了广泛的发展，已经成为应用服务器端的标准技术。

SUN EJB 技术是在 Java Bean 本地构件基础上，发展的面向服务器端分布应用构件技术。它基于 Java 语言，提供了基于 Java 二进制字节代码的重用方式。而且，EJB 给出了系统的服务器端分布构件规范，包括构件、构件容器的接口规范以及构件打包、构件配置等标准规范内容。因而，EJB 技术的推出，使得用 Java 基于构件方法开发服务器端分布式应用成为可能。

从企业应用多层结构的角度，EJB 是业务逻辑层的中间件技术，与 Java Bean 不同，它提供了事务处理的能力，自从 3 层结构提出以后，中间层，也就是业务逻辑层，是处理事务的核心，从数据存储层分离，取代了存储层的大部分地位。

从分布式计算的角度来看，EJB 像 CORBA 一样，提供了分布式技术的基础。提供了对象之间的通信手段。

从 Internet 技术应用的角度来看，EJB、Servlet 和 JSP 一起成为新一代应用服务器的技术标准，EJB 中的 Bean 可以分为会话 Bean 和实体 Bean，前者维护会话，后者处理事务，此时 Servlet 负责与客户端通信，访问 EJB，并把结果通过 JSP 产生页面传回客户端。

J2EE 的优点是，服务器市场的主流还是大型机和 Unix 平台。这意味着以 Java 开发构件，能够做到“Write once，run anywhere”，开发的应用可以配置到包括 Windows 平台在内的任何服务器端环境中去。

3.6.3　OMG 的 CORBA

CORBA 分布计算技术是 OMG 组织基于众多开放系统平台厂商提交的分布对象互操作内容的基础上制定的公共对象请求代理体系规范。

CORBA 分布计算技术，是由绝大多数分布计算平台厂商所支持和遵循的系统规范技术，具有模型完整、先进，独立于系统平台和开发语言，被支持程度广泛的特点，已逐渐成为分布计算技术的标准。也是一个用于开发和配置分布式应

用的服务器端中间件模型规范

目前，CORBA 兼容的分布计算产品层出不穷，其中有中间件厂商的 ORB 产品，如 BEAM3，IBM Component Broker；有分布对象厂商推出的产品，如 IONA Obix 和 OOCObacus，还有科研机构研制的 ORB 产品，如华盛顿大学的 TAO 等，我国主要从事 CORBA 研究的有国防科技大学与东南大学等单位，并有相应的产品。

CORBA 规范的近期发展，增加了面向 Internet 的特性，服务质量控制和 CORBA 构件模型。Internet 集成特性包括了针对 IIOP 传输的防火墙和可内部操作的 URL 命名格式的命名服务（naming service）。

服务质量控制包括能够具有质量控制的异步消息服务，一组针对嵌入系统的 CORBA 定义，一组关于实时 CORBA 与容错 CORBA 的请求方案。

CORBA 构件模型 CCM（CORBA Component Model）技术，是在支持 POA 的 CORBA 规范（版本 2.3 以后）基础上，并结合 EJB 当前的规范发展起来的。

CORBA 构件模型规范主要包括如下 3 项内容：

① 抽象构件模型，用以描述服务器端构件结构及构件间互操作的结构；

② 构件容器结构，用以提供通用的构件运行和管理环境，并支持对安全、事务、持久状态等系统服务的集成；

③ 构件的配置和打包规范。CCM 使用打包技术来管理构件的二进制、多语言版本的可执行代码和配置信息，并制定了构件包的具体内容和基于 XML 的文档内容标准。

如果遵循 CORBA/IIOP 开发对象应用，则在开发新应用时，便可以在程序中以 API 形式调用如安全服务、对象事务服务、并发服务、生命期服务等 CORBA 公共服务。

3.6.4 三种技术支持下的分布式构件技术

目前，针对上述的各种分布计算平台技术，都出现了相似且具有可比性的分布式构件，即 CORBA CCM 技术、SUN 的 EJB 技术和 DNA 2000 中的 COM/DCOM/COM+ 技术。其中，COM 技术和 EJB 技术已得到较为广泛的应用，CCM 则是在继承和吸收 EJB 当前规范的基础上，基于 CORBA 规范制定的服务器构件应用开发模型。由于 OMG 组织的 CORBA 规范一直为广大开放系统平台厂商所支持，使得 CCM 规范也具有既不局限于特定系统平台，也不局限于特定开发语言的特点，具有广泛的兼容性。因此，具有强大的生命力。

上述分布计算平台技术由于产生的历史原因和技术不同，在应用上也具有差别。其中，J2EE 技术下的 EJB 局限于 Java 编程语言，并且由于 Java 应用必须在 Java 虚拟机中运行，因此这一方面限制了 Java 对已有遗留系统的兼容性，也使

得其性能有所降低；DNA 技术则局限于微软的操作平台，因此对于开放系统平台厂商，如 Unix 和 Linux，则不能提供支持。

目前，CORBA CCM 规范已发布，爱尔兰的 IONA 公司、美国的 DEVSYS 公司和华盛顿大学的 TAO 项目、法国 UST 大学的 ECOOP 2000 项目均正在研制相关的原形系统。

有关这 3 种分布式组件模型的细节阐述参见本书的第 4 章。

3.7　中间件的集成和应用

3.7.1　中间件在网站系统中的集成应用

随着计算机技术的快速发展，广大计算机用户不再满足于简单的客户/服务器这样的两层结构系统，不断提出应用系统 3 层甚至多层体系结构的需求。所谓“3 层”，就是在原有的 2 层结构（客户端和服务器端）之间增加了 1 层应用，将原先客户/服务器结构中由客户端实现的部分功能剥离出来，放到中间这一层来实现。同时，由于大规模应用系统软件也通常要求在软硬件平台各不相同的网络上运行，所以一种基于标准的又独立于计算机硬件以及操作系统的中间件就应运而生了。顾名思义，中间件就是处于应用软件和系统软件之间的一类软件，它独立于硬件厂商或数据库厂商，是客户方与服务方之间的连接件，是需要进行二次开发的中间产品。一般认为，中间件是一种独立的系统软件或服务程序，应用它可以在不同的硬件平台之间实现资源共享。中间件置于客户机服务器的操作系统上，管理计算资源和网络通信。中间件的突出特点一是网络的通信功能，它不仅要实现互连，还要实现数据之间的相互交换和应用之间的相互操作；二是能使用户可以不再关心整体系统中硬件平台、操作系统、数据库这些底层应用环境的不同，而将注意力集中在应用逻辑处理的开发上面。

目前虽然中间件的产品种类很多，但根据它在系统中所起的作用和采用的技术不同，大致可划分为远程过程调用中间件、面向消息中间件、数据库中间件、对象请求代理中间件和交易处理中间件 5 种。从目前的发展趋势看，这 5 种类型有相互融合的可能。最近几年，交易处理中间件这一类型的产品在众多领域中被广泛应用，所以得到了迅速发展。目前，在国内外交易处理中间件产品中，比较优秀的产品很多，如美国 BEA 公司的 Tuxedo、IBM 的 CICS、日本日立的 TOPEND、我国北京东方通科技发展有限责任公司的 Tong Easy 等。这些产品都具备了成熟性、稳定性、可靠性、生命力强、性能优秀的特点。这些产品简单易用，应用开发人员能够很快地掌握它，并能利用它提高开发的质量和速度。值得一提的是，在上述 4 家厂商的中间件产品中以美国 BEA 公司的 Tuxedo 最为引人注目。Tuxedo 是 1984 年在贝尔实验室开发成功的，已被全球 2500 多家大型机

构的关键任务应用系统所采用，其中多数是世界级企业的应用系统。该产品涉及金融、电信、交通、零售、制造、医疗、政府等领域，在中间件市场中占有很大的份额，且发展前景良好。

互联网发展至今，传统网站系统的2层体系结构在实际运用中已经暴露出了不少问题。如程序的业务逻辑处理一般存在于前台的应用程序中，因而程序没有通用性。一旦客户的业务逻辑有所改变，相应地需要修改应用程序，将所有程序模块重新进行修改、编译、连接，其工作量相当大；还有，由于客户机直接访问数据库，也不利于安全控制，难以防止黑客的恶意攻击；另外，传统网站系统的2层结构在维护和扩展方面也存在着较大难度。中间件的采用，把传统网站系统的2层体系结构转变成了3层甚至多层体系结构，不仅使网站系统的性能、安全性、扩展性方面有了很大的提高，而且也方便网站系统的维护和管理。

利用中间件构建网站系统的优势在于：

① 提高了网站系统的性能。中间件可以对服务进程进行管理，使一个服务进程处理多个网站用户的访问请求，而不再像传统网站2层体系结构中的一个服务进程只能处理一个访问请求。当访问请求的数量超过了服务器的处理极限时，中间件还会利用队列缓冲机制把服务器来不及处理的请求放入队列中，等服务器空闲时再进行处理。

② 提高了数据库响应速度。通过中间件，服务进程可以与数据库一直保持连接，当用户访问网站系统时大大减少了与数据库连接的次数和时间，有效地解决了因为数据库连接数过多引起数据库性能下降的问题。

③ 提高应用系统的安全性。中间件将客户端与数据库隔离起来，客户端无权直接访问数据库，有利于安全管理，可有效防止恶意攻击。还可以利用中间件的安全管理特性进一步加强权限控制管理。

④ 提高了网站系统的扩展能力。采用中间件后，若要提高系统性能、处理速度，可增加中间层的应用服务器，分担一部分应用服务工作即可。

由于上述优势，网站系统将会越来越多地采用中间件，有可能形成一种利用中间件构建网站系统的技术发展趋势，从而使网站系统更出色地发挥其性能。

3.7.2 大规模软件构架中的中间件集成框架

大规模的软件构架中，需要各种中间件的支持才能彻底解决异构问题，并且使这种大规模软件的构架更有效。为了适应大规模的软件构架的需求，未来的中间件应以集成框架的形式出现，其体系结构如图3-17所示。

框架（framework）是一个软件环境，用来简化应用开发和系统管理。框架位于中间件服务之上，是中间件服务的使用者。同样，位于框架之上的应用软件是框架的使用者，是中间件服务的间接使用者。由于中间件服务的广泛性和复杂

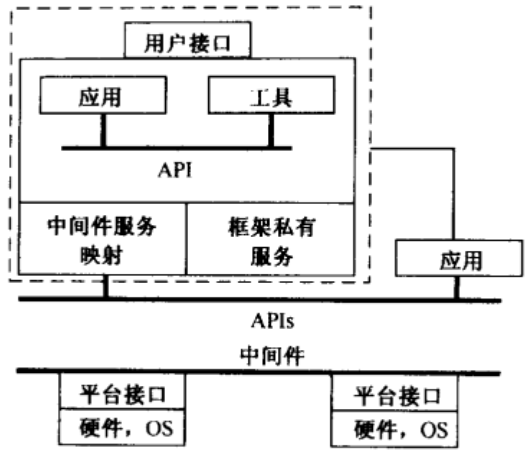


图 3-17　中间件集成框架结构

性，越来越多的应用软件开始依靠框架来简化下层软件环境，而不再直接访问中间件服务。

框架的 API 可以是中间件服务 API 的简单汇总。框架可以是一个封装下层服务的外壳。对用户接口进行专门化、通过共享上下文来简化 API 和增加框架私有的中间件服务等是框架的主要功能。

框架可以有自己的用户接口，对下层平台有专门化的 GUI。例如，一个办公系统框架可以把 GUI 专门化，提供类似办公桌的界面，带有适合办公室用户的图标和排放方式。框架可通过维护跨越调用的上下文来简化其 API。例如，服务调用要求有一个用户标识和一个设备标识作为参数。然而，框架可以维护这样的标识作为应用的上下文，不再要求它们作为参数，从而简化了 API。

框架也提供私有的中间件服务，这通常是由于框架需求某种服务，而标准的中间件却不提供。例如，CASE 框架常提供专用的服务用于版本和配置管理。这些服务在将来很可能被标准的中间件所取代。

框架通常包括各种工具，它们是一些通用的应用软件，使得框架功能更强、更容易使用。工具可以面向终端用户、程序员和系统管理员，可以由框架开发者提供，也可以由第三方提供。常见的工具有编辑器、编译器、调试器、性能监视器、软件安装管理器等。

如果工具通过数据、控制、表示与框架集成，它就成为框架的一部分。在数据集成的情况下，一个工具与同为框架一部分的其他工具共享数据（一般是在外存上的数据），这就要求工具之间在共享数据的格式和操作方式上达成一致。有时格式由数据的拥有者定义，其他工具可以引用；有时数据由多个工具对等地共享。在控制集成的情况下，框架中的一个工具能够调用另一个工具，被调者可以

与调用者交换适当的数据和控制信息。在表示集成中，多个工具具有相同的外观，通常使用通用的 GUI，采用标准的使用规范（如剪切、粘贴、拖、放）。有时表示集成会促进对数据集成和控制集成的需求。比如，如果从复合文档编辑器中调用电子表格，那么数据集成（包含电子表格的内容）和控制集成（加载电子表格应用）二者缺一不可，但只有通过表示集成才允许用户表达这种逻辑关系

表示集成和控制集成通常比数据集成更易于实现。对于控制集成，每个工具定义自己接口，并提供通过中间件通信服务对这些接口的访问，从而可以独立地集成。这项工作集中在工具的接口，不要求对工具具体有任何修改。对于表示集成情况类似，而数据集成则要求所有工具对所共享的数据格式和存在方式达成一致。由于数据访问可能遍布工具的各部分，因此数据格式的改变会引发大量繁杂的工作。

由于中间件技术的日趋完善，为软件的体系结构注入了新鲜的活力。目前给一套中间件服务增值的重要途径是把它们集成在一起，从而使它们像一个统一的系统一样工作。这些集成了的服务常常封装在一个框架内，而大规模软件的构架迫切需要中间件走这种集成之路。

参考文献

常熠芬，张育平.2001. 中间件技术研究.计算机应用研究，10(10)：21～23

甘卫东，张开德.2001.CORBA 实时性研究初谈.计算机工程与应用，1(1)：73～75

胡玲园. 1999. 关于客户机/服务器中间件技术的研究.上海电力学院学报，14(2)：12～17

胡雅庆.2001. 面向消息中间件设计与实现.计算机与现代化，73(3)：41～45

李立宏等.2000. 消息中间件的设计与实现.计算机工程，26(1)：46～47

林淘等.2001. 用通信中间件实现 B2B 与 ERP 的系统集成.计算机应用，21(5)：24～26

娄渊胜，尹燕敏，王志坚.2001. 基于 CORBA 的通用数据存取中间件研究及实现.小型微型计算机系统，22(10)：1210～1212

马松，盛浩林.1999.Internet 上数据库中间件原型的研究与构造.软件学报，10(1)：86～89

宋晓梁，刘东生，许满武.1999. 中间件及其在三层客户机/服务器模型的应用.计算机应用，19(7)：35～38

王辉，施小英.1999. 中间件服务及其集成框架.计算机工程与应用，9(9)：25～27

王飞杰，纪晏宁，余柏华.2001. 基于中间件的客户/多服务协作模型的研究及应用.计算机应用，21(6)：7～9

王晓东，彭兵，张际平.2001. 基于中间件的开发研究.计算机应用研究，14(10)：54～57

徐金梧译.2000. 基于C++的 CORBA 高级编程.北京：清华大学出版社

杨立平等.2001. 数据库中间件技术及在三层客户机/服务器模型中的实现.小型微型计算机系统，22(4)：482～484

张岩，周可记.2001. 中间件技术与应用研究.计算机与通信，1(1)：31～38

张伟华，祁新.1999. 中间件在远程调用技术的研究及在 TeeSVR 中的应用.上海铁道大学学报，

20(10):49～53

张下梅. 2001. 服务器端中间件技术.计算技术与自动化,20(1):75～78

郑雪,徐亚娟.1999. 中间件的概念、分类和应用.微电脑应用,2(2):15～17

OMG 著.2000.CORBA 系统结构、原理与规范. 北京:电子工业出版社

Box D.1997.Q&A ActiveX/COM. Microsoft Systems Journal, (3):93～105

Bruce Robertson.1993.Making Sense of Middleware. Network Computer, 20(10):54～60

Cynthia McFall.1998.An Object Infrastructure for Internet Middleware. IEEE Internet Computing, 2 (2):102～110

Ed Roman.1999.Mastering Enterperise Java Beans and the Platform, Enterprise Edition. British:John Wiley & Sons, Inc.

OMG.1994.The common object request broker:architecture and specification, revision 2

OMG.1997.CORBA2.1 UpdateSheet

OMG.1999.http://www.omg.org

RonBen-Natan.1995.CORBA-A Guide to Common Object Request Broker Architecture. New York: McGraw-Hill

第 4 章　构件与构件模型技术

随着软件工程和面向对象技术的发展，应用程度从基本的数据结构到子程序到模块到程序库到类和抽象数据类型，直到最新的构件（component），其抽象程度越来越高。构件模型通常由基于各种语言开发工具、构件嵌入机制和相关服务（事务、安全、认证、负载均衡、生命周期等）组成。可以利用脚本语言（Script）将各构件集成一个应用系统。目前比较成熟的构件模型有 OMG 的 CORBA，SUN 的 EJB 和微软的 ActiveX/COM/DCOM/COM + 。

4.1　CORBA 构件模型 CCM

4.1.1　CORBA 概述

CORBA 是对象管理组织 OMG 的一个规范，它的底层和核心部分是对象请求代理（ORB）。从某种意义上说，CORBA 是“软件总线”。CORBA 的实质是 RPC 与面向对象技术的有机结合。在 CORBA 中，每一个构件是一个对象，有一个基于面向对象的接口，内部代码实现可以是 OO 或非 OO 的语言，总线上的对象能够被任何其他对象所使用。CORBA 提供了接口定义语言（IDL）到 C、C++ 、Java、Cobol 等语言的映射机制，其具体实现者是 IDL 编译器。IDL 通过编译可以生成服务器方 Skeleton 代码和客户方 Stub 代码，通过分别与客户端和服务器端的联编即可得到服务方和客户方的程序。对一个 CORBA 客户机来说，服务器的位置是透明的，客户机通过接口与服务器构件进行通信，这些接口是用 IDL 来定义的。

CORBA 同时提供了一系列的公共服务规范，其中包括名字服务、永久对象服务、生命周期服务、事务处理服务、对象事件服务和安全服务等，它们相当于一类用于企业级计算的公共构件。CORBA 的对象服务（object services）定义了那些在对象服务中是最重要的服务。目前已定义的服务包括命名服务、对象生命周期服务、安全服务、属性服务、并发服务、查询服务、时间服务等。随着应用的发展，还得定义更多的对象服务。

CORBA3.0 作为一个全新的标准，融合了许多新的技术，并在充分考虑了互操作性和可移植性的基础上，完善了原有模型的功能，扩展了应用领域，为推广分布式应用打下了良好的基础。

4.1.2　CORBA 的组成及体系结构

最新的 CORBA 规范主要包含以下内容：ORB 核心、OMG 接口定义语言、语言映射、存根（Stub）和框架（Skeleton）、动态调用和仓库、对象适配器等。图 4-1 描述了 CORBA 的主要组成部分和它们之间的关系。下面对每个构件作较详细的论述。

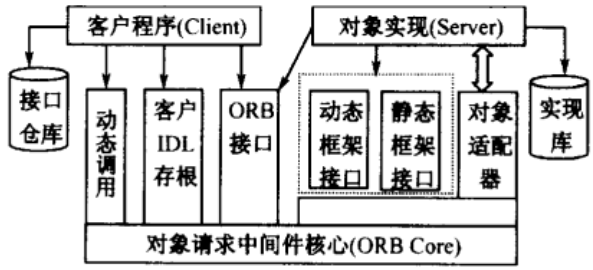


图 4-1　CORBA 的主要组成部分

1. ORB 核心

ORB 的任务是，把客户发出的请求传给目标对象，并把目标对象的执行结果返回给发出请求的客户。由此看出，ORB 的最重要的特征是，提供了客户和目标对象之间的交互透明性。具体地说，它主要屏蔽了以下内容：对象位置、对象实现的细节、对象的执行状态、对象通信机制和数据表示等。ORB 的这些特点，使应用开发者不必过多地担心底层细节对分布式编程的影响，从而可以集中精力设计自己的具体应用。

2. IDL 语言

在客户向目标对象发送请求之前，它必须知道目标对象所支持的操作的特性。对象的界面就是用来说明对象所支持的操作类型。CORBA 的对象界面由 OMG IDL 来定义。OMG IDL 的语法与C++ 类似（包括C++ 的预处理语句），它另外增加了一些支持分布式处理的关键字（如 in 和 out 等）。IDL 为纯粹的说明性语言，只对对象的界面（包括异常处理）进行定义，不提供对象的实现细节，不能用它直接去实现分布式应用。对象的实现由主持 CORBA 的语言完成，并且可在任何支持 CORBA 的语言中调用。CORBA 规定了从 IDL 到程序语言的映射方式，这种映射的实现交给开发商完成。目前支持 CORBA 的语言包括 Ada、C++ 、Cobol、Smalltalk、Java、C 等，用 IDL 统一描述对象界面的操作性和语言的独立性。IDL 并不包括循环、分支等控制结构。OMG IDL 编译器除了

把 IDL 的特性映射到具体的编程语言之外，还根据界面描述来产生客户方的存根和服务方的框架（也有人称之为服务方存根）。存根代表客户创建并发出请求；框架则把相应的请求交给 CORBA 对象实现。由于存根和框架都是通过对用户的界面定义进行编译而得，所以通过存根和框架的调用被通称为静态调用。

在图 4-1 中，Client 和 Server 对象均用 IDL 语言描述。IDL 通过指定对象接口来定义这些对象的类型。一个接口包括一组已命名的操作和这些操作的参数，尽管 IDL 提供了描述 ORB 所操纵对象的概念性框架，但 ORB 正常工作并不必要求 IDL 源代码，只要等价的信息符合存根例程或实时接口库的格式，一个特定的 ORB 就能正确地实现其功能。当然，对于不同的对象系统，应提供一套 IDL 的映射机制，这实际上是 IDL 语言的编译器。IDL 的基本语法为：

```
<specification>::=<definition> +
<definition>::=<type-def>";"|<const-def>";"|<except-def>";"
|<interface>";"|<module>";"
```

其中，

“::=”表示“被定义为”；

“|”表示“任选”；

“+”表示“该语法单位可以被重复一至多次”。

IDL 支持基本数据类型，如基本类型中的字符型、整型和浮点型等，以及构造类型中的数组、枚举、联合、结构等和模板类型中的列表、字符串等。另外，还有一个 any 类型可支持任何数据类型。

CORBA IDL 说明对象的属性、所继承的父类（允许多继承）、运行时可能导致的异常、引发的事件以及对象提供方法（包括输入参数和返回结果的类型）。IDL 的主要结构是接口，类似C++ 和 Smalltalk 等的类接口，因而很容易映射到这些语言的对应结构上。IDL 定义的接口在分布式对象（构件）的实现者和调用者之间建立起 API 界面。所有的接口保存在 CORBA 界面仓库中，供对象在运行时查询和构造动态调用。

IDL 编译器把对象接口（假定为 xxx.IDL）映射成某种语言（如C++），所产生的文件中有两个重要文件 xxxS.X 和 xxxC.X（如果映射成 C++ 时，则分别为 xxxS.C 和 xxxC.C），前者为用于服务器端 Skeleton 代码，通过 BOAimpl 方法或 Tie，用C++ 语言实现服务类；后者作为用于客户端的 Stub，说明远程对象的本地代理，通过该代理对象，客户程序像访问本地对象一样访问远程对象。如图 4-2 所示。

作为 CORBA IIOP 的纯 Java 解决方案，Netscape 和 Visigentic 的 Caffeine 提供了 Java2IDL 编译器，能够由 Java 代码自动生成 CORBA 的 IDL，从而使 IDL 对 Java 程序员完全透明。

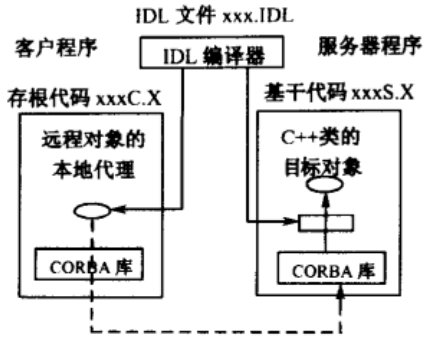


图 4-2　IDL 编译器的工作流程

不同的面向对象语言和非面向对象语言可以以不同的方式访问 CORBA 对象。对于面向对象语言而言，它希望看到的是对象的形式，即使对非面向对象语言来说，它所希望看到的也不包括具体的内部实现。将 IDL 映射为编程语言的方法对于所有的 ORB 实现应该是一致的。这些映射可能包括数据类型的映射和调用 ORB 的过程（或函数）接口的映射。语言映射还定义了对象调用和客户实现中的控制线程之间的相互作用。最普通的映射提供了同步调用，结果可以在过程完成时返回。其他的映射可以用来初始化调用并将控制权返回给程序，在这些情况下，附加的函数必须有相应的同步功能。

为了映射非面向对象语言，将有一个针对每个接口类型的程序接口。通常，句柄将提供访问 IDL 定义的操作机制。句柄调用对于 ORB 核心是私有的那部分 ORB。如果有多于一个 ORB，将会有对应于不同 ORB 的接口。在这种情况下，需要 ORB 和语言映射相互协调以访问正确的对象引用句柄。面向对象语言不需要句柄接口。接口允许对象动态调用，用户可以不必调用一个特定对象上的操作，它可以指定调用特定的对象，客户程序提供关于操作和参数类型的信息就可以了。

3. ORB Client

(1) IDL 存根和动态调用 API

Client IDL 存根提供了访问对象服务的静态接口，这些由 IDL 编译器生成的预编译存根，对应于 Client 如何激活处于 Server 上的相应服务；另一方面，通过使用 DII 提供的界面，客户可以动态地激活 API 而实时地发现 Client 想发现的服务对象，并调用其方法。CORBA 规定了描述对象界面元数据的方式和标准的 API、发现对象、构造参数、发出远程调用以及取得返回结果。与静态调用相比，DII 允许应用程序在运行时访问在编译该应用程序时未知的对象。

(2) 接口仓库 IR (Interface Repository)

接口仓库 API 允许用户获取和修改所有已注册构件接口的描述，它们支持的方法及这些方法所需的参数，称之为方法签名。IR 负责存储、修改和管理对象接口定义，用户程序使用 IR API 存取和修改这些信息。接口库提供接口定义的存储管理，并提供对用 IDL 描述的对象定义集的存取，IR 中的信息可以被 Client 和 Server 共同使用。IR 中接口定义通常具有以下特征：

① 包含该接口支持的操作描述，操作参数类型，该操作可能产生的异常及它所涉及到的上下文；

② 存储常数值（constant value），这些值可能在其他接口定义中被使用或仅为编程方便而定义；

③ 存储类型码（type code），类型码是用结构化术语描述的类型的值；

④ 使用模块（module）作为对接口进行分组的方法和作为对按名的分组进行导向的方法。

IR 中的接口具有名字和标识号（IDs）。IR 中每一个接口被作为一组接口对象来维护，分别为 Repository（库名空间的顶层模型）、Module Def（接口的逻辑分组）；Interface Def（接口定义）、Attribute Def（接口的属性定义）、Operation Def（接口中操作的定义）、Typedef Def（不属于 Interface 类型的已命名类型所定义的基本接口）、Constant Def（已命名的 Constant 定义）和 Exception Def（一个操作所引起的异常的定义）。

对于 IR 中的接口对象，都可以由几个基本接口来定义：IRObject、Container、Contained、IDLType 和 Typedef。这些基本接口都有各自的定义，下面给出了 IRObject 接口的定义，IRObject 接口提供了标识对象实际类型的操作。

```
module CORBA {
    interface IRObject {
        read only attribute DefinitionKind derf-kind; //读接口
        void destroy (); //写接口
    };
};
```

说明：

destroy 操作引起对象中止并退出；

DefinitionKind 是 enum 类型，它在下面 module 中定义：

```
module CORBA {
    typedef string Identifier;
    typedef string ScopedName;
    typedef string RepositoryId;
    enum DefinitionKind {
        dk-none, dk-all,
```

```
            dk-Attribute, dk-Constant, dk-Exception, dk-Interface,
            dk-Module, dk-Operation, dk-Typedef,
            dk-Alias, dk-Struct, dk-Union, dk-Enum,
            dk-Primitive, dk-String, dk-Sequence, dk-Array
        };
    };
```

定义了基本接口就可以用基本接口来定义上面的接口对象，以 Repository 接口为例，它的定义如下：

```
module CORBA {
  interface Repository: Container {
    Contained lookup-id (in RepositoryId Search-id); //读接口
    Primitive Def get-primitive (in PrimitiveKind kind);
    String Def create-string (in unsigned long bound); 写接口
    Sequence Def create-sequence {
        in unsigned long bound;
        in IDL Type element-type;
    };
    Array Def create-array {
        in unsigned long length;
        in IDL Type element-type;
    };
  };
};
```

说明：

//读接口

lookup-id 操作：查找给定库中对象的库 ID；

get-primitive 操作：返回一个具有指定类型属性的对 Primitive Def 的一个引用。

//写接口

create-〈type〉操作：创建一个用来定义隐名（anonymous）类型的新对象，其中隐名类型是指 String、Sequence 和 Array。

(3) ORB 接口

ORB 接口是一种直接对应于 ORB 的接口，它对于所有的对象接口，对象适配器都是一样的。大部的操作都由对象适配器、句柄、框架或动态调用实现，因而 ORB 接口仅仅是所有对象间的一些公共操作。接口库是一种服务，其中保存着接口信息，这些信息在 ORB 执行请求时会用上。而且，当一个应用程序在调用一个未知接口的对象时，可以通过接口库了解能够在其上进行的操作。Client

和 Server 根据各自的语言映射拥有对象引用的真实意图，对象引用提供了在基于 ORB 的 Client/Server 系统中惟一标识一个对象所需的信息，它具有惟一的名字和标识。因此它们与对象引用的实际表示相隔离。两个 ORB 实现可能选择不同的对象引用来表示，这样就允许用特定语言编写的程序存取与特定 ORB 无关的对象引用。

4.ORB Server

由于静态和动态激活对象引用均提供了相同的激活消息语义，所以 Server 并不知道这两种引用方式的区别。ORB 通过 Server 的 IDL 存根（或称框架）定位一个对象适配器，将控制参数和转换控制传输给对象实现。

（1）静态框架（static skeleton）

静态框架在 Server 端，Server 端输出的每个服务通过此接口。这些接口像 Client 存根一样，使用 IDL 编译器创建。

由于 Skeleton 和 Stub 是由 IDL 编译器编译连接生成的，分别用于客户端和服务器端，用来完成对象远程调用时参数和返回结果数据的包装和取出，并且使结构型数据的传输对程序员透明。客户程序对服务程序的调用是在编译时确定的，因此称静态调用。实现服务的方法有 ImplBase 方法和 Tie 方法。ImplBase 是通过继承由 IDL 编译器生成的 Skeleton 代码来构造服务类，实现 IDL 接口定义的操作和属性；Tie 方法则直接构造服务类，并产生服务类的一个接口类（称 TIE 类），客户对象通过 TIE 对象间接访问服务对象。

静态调用限定客户程序只能使用编译时能够确定的 IDL 接口。服务器通过回调（callback）可以访问客户端的对象（这时客户端转换为服务器）。

CORBA 静态方法调用步骤如下：

① 用 IDL 定义对象类。IDL 定义了对象的类型、对象的属性、对象输出的方法以及方法的参数。

② 通过语言预编译程序运行和处理 IDL 文件，为实现服务器类产生语言框架。

③ 为生成的框架提供实现方法的代码。

④ 编译代码，产生三类输出文件：一是向接口仓库描述对象的输入文件；二是 IDL 定义的方法的客户存根——需要通过 ORB 静态访问 IDL 定义的服务的客户程序将调用这些存根；三是在服务器上调用这些方法的服务器框架。

⑤ 将类定义与接口存储库联编。

⑥ 向实现仓库注册运行对象。对象适配器在实现仓库里记录对象标记以及在服务器上实例化的对象类型。实现仓库还知道哪一些对象类在某个特定服务器上得到了支持。对象请求中介利用这个信息去查找某个特定服务器上的活动对象

或者请求激活对象。

⑦ 在服务器上实例化对象。在启动时，服务器对象适配器可以将远程客户端方法调用服务的服务器对象实例化。

(2) 动态框架 (dynamic skeleton)

动态框架接口 DSI (Dynamic Skeleton Interface) 为服务程序提供了运行时的捆绑机制，它检查所接受的消息参数以决定目标对象和方法。

允许动态处理对象调用的接口是非常有用的，不是由与特殊操作相关的框架来访问对象实现，而是由一个提供访问操作名和参数的界面，用一种类似于动态调用接口的方式来访问对象实现。动态框架界面可以由客户句柄或动态调用接口来调用，它们向动态框架接口发出对象请求。动态框架接口的基本思想是让所有的对象请求通过调用同一组例程来达到调用对象实现中方法的目的。

在服务方，接收到调用请求的服务对象不需要了解该请求是通过 SSI 或者 DII 发出的。DSI 是服务方与 DII 对应的部分，它允许服务器接收对任何对象的操作或属性调用，甚至是编译时未知其 IDL 接口的对象。并且服务不必挂接到一个接口 Skeleton 代码上，就可以对该对象进行访问。客户程序不知道服务器是用 DSI 实现的。为做到这一点，服务器定义一个函数，当有操作或属性到来时，调用该函数，由它判断被调用的对象、操作的名字以及参数的类型和值，然后完成客户程序请求的任务，并构造返回结果。DSI 帮助设计 CORBA 和非 CORBA 之间接口的网关程序，而 DII 则可以构造从非 CORBA 到 CORBA 的网关程序。

动态调用常用到两个接口：TypeCode 接口用于在运行时描述对任意复杂的 IDL 类型；Any 接口可以支持传递任何类型的参数和返回值。DII 和 DSI 是真正系统级的 API，可以动态包装和取出调用请求，并且它们是相互独立的，基于静态调用的客户程序可以通过 DSI 访问服务对象，基于 DII 的客户程序也可以通过 Skeleton 代码访问服务器对象，对客户程序透明。

动态调用的大致步骤如下：

①获得接口名字。CORBA 对象是自省的，它们能提供相当多有关自己的信息。因此，可以通过调用 GetInterface()方法向这个对象询问其接口的名字。这个调用将返回一个描述该接口的对象。

②从接口仓库获得方法描述。可以把刚才返回的接口的对象用作实现接口仓库导航的进入点。客户可以通过 LookName()方法去寻找它想调用的方法，然后再通过 Describe()方法去获得这个方法的全部 IDL 定义。

③创建变元列表，CORBA 规定了一个自定义的数据结构以传递参数，这叫做命名数值列表 NVList。用户可以使用 NVList 准对象实现这个列表。通过调用 CreateList()方法创建该列表，和很多 AddItem()调用一样，它要将每个变元增加到列表上。

④创建请求。请求是一个包含方法名、变元列表和返回值的 CORBA 准对象。可以通过调用 CreateRequest( )方法来创建请求。必须将要调用的方法名、NVList 准对象和返回值指针传递给这个方法。

⑤调用远程方法。对于 CORBA 对象请求中介的静态和动态调用，客户通过访问对象标记（ObjectID）和调用执行这个服务的方法来发出请求。服务器不能区别静态和动态调用。对象实现、对象适配器以及用来访问这个实现的对象请求中介，对静态和动态客户是完全透明的。

（3）对象适配器 OA（Object Adapter）

对象适配器是对象访问 ORB 提供的服务的主要方式。由 ORB 提供的服务在一个对象适配器中经常包括：对象引用的产生和解释，方式调用，交互性安全，对象和实现的激活与释放，对象引用到实现的映射及实现的定位，各个不同对象的粒度，生命周期等。ORB 内核无法为所有的对象提供一个统一、方便有效的界面。通过对象适配器的作用，可以将目的对象分成若干组，每组通过特定的对象适配器来满足其特定的需要。但这样一来，对象适配器的种类便会急剧膨胀，为了减少对象适配器的种类，CORBA 给出了基本对象适配器（BOA），以满足大多数对象的需要，BOA 提供了产生和解释对象引用、对请求进行认证、激活/去活实现 、激活/去活单个对象、通过框架调用方法等功能。在提供这些功能时，BOA 要用到一些与操作系统有关的知识，这些知识由实现仓库提供，实现仓库还存放对象实现的有关信息。

对象适配器是对象位于 ORB 核心通信服务的顶层，负责接受代表 Server 对象的服务请求。它为实例化 Server 对象、向 Server 传送请求和为 Server 指定对象引用的对象 ID 提供一个实时环境。对象适配器也能注册它们所支持的类和实现库中的实现的实时实例（即对象）。对象适配器的结构如图 4-3 所示。

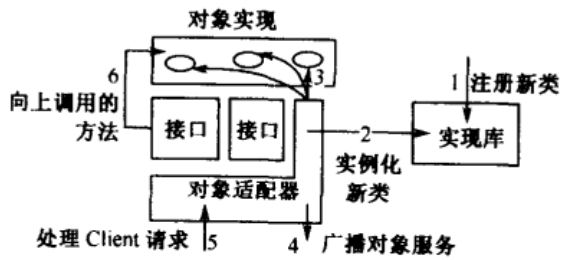


图 4-3 OA 的结构

另外，Server 可以支持多个对象适配器，因为一个对象适配器定义了在当前进程或线程中是否激活一个对象方法，是否创建一个新的进程，是否创建一个线程。BOA（Basic Object Adapter）是 ORB 必须支持的标准适配器。BOA 引起服

务器和不同的 ORB 接口的不一致性，给服务器程序在不同的 ORB 间的移植带来了困难。CORBA2.2 引进可移植版本 POA（Portable Object Adapter），它规定可移植的 API。POA 还支持临时兑现和持久对象，并允许为接口的类实现有选择地提供一个侍从管理员（servant management），通过向自己的 POA 注册管理员，允许服务器控制服务对象的创建、激活和撤消等。POA 是 ORB 的重要组成部分，是介于 ORB 核心和服务器程序之间的软件层，它把所有的 CORBA 对象进行分类管理，将客户端发来的请求正确、快速地调度到其对应的目标对象上进行操作并返回结果。OMG 提供了 7 种策略来控制 POA 的特性，它们是生存期策略、对象标识符分配策略、标识符惟一策略、隐式激活策略、请求处理策略、伺服程序保留策略和线程策略。根据不同的应用可以有不同的策略组合，然而策略选择的好坏将直接影响到 POA 的性能，进而影响到整个 CORBA 平台性能的发挥。

在客户程序看来，所有的服务对象都是活动的，并等待客户程序发出调用请求。当有成千上万个服务对象时，系统无法做到这一点，ORB 的服务器方仅给客户程序提供了这样一种假象。程序员编写服务程序时，必须帮助 ORB 提供自动的启动功能，ORB 能够自动启动一个对象。接口的类实现也必须同永久服务合作来保存和恢复服务对象的状态。给客户程序提供的假象最终是由服务对象和 BOA 协力完成的。OA 激活对象的方式是，或者创建一个进程、或者在现有的进程内创建一个线程，也可以利用现有的线程或进程管理对象。服务器可支持多个 OA 满足不同类型的请求。

自 CORBA2.0 开始，对服务器和服务对象作了明确的区分。服务器是执行单元，即进程；对象实现了一个接口，一个服务器包含一个或多个对象，对象总是在它们的服务器内被激活，并在 CORBA2.0 中定义了以下的激活策略：

① 共享服务器（shared server）。具有相同服务器名的对象由同一个进程管理，当对服务器包含的对象的请求首次到达时，BOA 激活该服务器。

② 非共享服务器（unshared server）。服务器的每一个对象由单独的进程来管理。

③ 每个方法一个服务器（server－per－method）。每当 BOA 收到一个调用请求，就创建一个进程，操作完成时进程结束。

④ 永久服务器（persistent server）。服务器由 BOA 之外的手段激活，比如，手工激活服务器程序，然后在程序中通知 BOA，准备接收外部请求。

（4）实现仓库 IR（Implementation Repository）

实现仓库包含了 Server 支持的类、被实例化的对象和对象标识 Ids。这些信息供给 ORB 定位和激活对象实现。尽管 IR 中的绝大多数信息对 ORB 或操作环境而言是特定的，但实现库是记录这些信息的通常场所。通常情况下，对象实现的安装和对象实现的激活与执行相关机制的控制均是通过作用于实现库上的操作

来完成的。同时，对于实现 ORB 功能而言，实现库也是存储与 ORB 的实现有关的附加信息的一般场所。如调试信息、管理控制、资源定位和安全等也与实现库有关。

5.ORB 之间的互操作

在发布 CORBA2.0 之前，ORB 产品的最大缺点是，不同供应商所提供的 ORB 产品之间并不能互操作。为此，CORBA2.0 给出了一个通用的互操作体系结构。

通过对象请求中介通信协议 GIOP（Global Internet－ORB Protocol）体系结构的基础是为 ORB 之间的通信规定了传输文法和信息格式。设计 GIOP 的原则是简单且易于实现。对任何面向连接的传输协议作极少量的假设后，其上都可以直接建立 GIOP。IIOP 说明如何在 TCP/IP 协议上建立 GIOP。IIOP 协议规定了 GIOP 消息如何在 TCP/IP 网络上交换，即可以看作 GIOP 的 TCP/IP 版本。

IIOP 出现之前，CORBA 只为同一平台上的对象提供互操作性，有了 IIOP 协议，CORBA 就可以实现跨平台间的互操作。这样就可以将 Internet 本身作为骨干对象请求中介，其他对象请求中介可以通过这个骨干请求中介来建立桥梁，从而使分布式对象与万维网之间找到理想的连接方式。

6.CORBA 的体系结构

根据以上的分析，CORBA 体系结构可进一步描述为如图 4-4 所示。

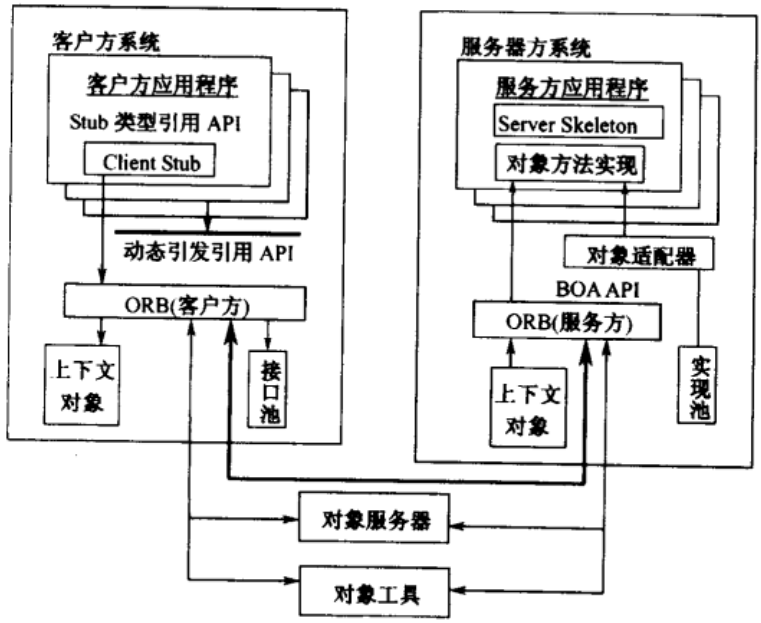


图 4-4 CORBAW 体系结构

4.1.3　对象管理体系结构 OMA

CORBA 是基于对象管理体系结构 OMA（Object Management Architecture）的，OMA 为构建分布式应用定义了非常广泛的服务。OMA 服务可划分为 3 层，分别称为 CORBAservices（CORBA 服务）、CORBAfacilities（CORBA 工具）和 ApplicationObjects（应用对象）。当应用程序需访问这些服务时，就需要 ORB 通信框架。这些服务在 OMA 中实际上是不同种类的对象的定义，并且为了支持分布式应用，定义了很广泛的功能。如图 4-5 所示。

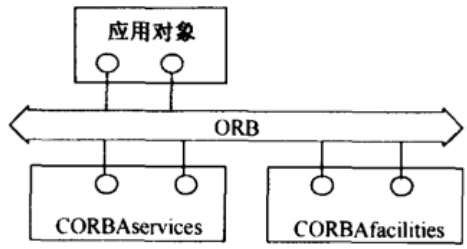


图 4-5　对象管理体系 OMA

CORBAservices 是用 IDL 说明的接口组成的系统级服务的集合，是开发分布式应用所必需的模块。这些服务提供异步事件管理，事务、持续、并发、名字、关系和生存周期的管理。CORBAservices 对 ORB 功能进行扩充和补充，包括如下几个方面：

① 与分布式系统相关的服务，如命名服务、事件服务、安全服务和交易服务；

② 与数据库相关的服务，如事务服务、特征服务、关系服务、查询服务、持久对象服务和外部化服务；

③ 一般服务，如生命周期服务、许可服务和时间服务。

这些服务并不同等重要，在不同领域其重要性也有区别。下面就几个重要的服务予以说明。

命名服务：允许给对象分配名字，之后，对象可以通过名字访问对象。命名服务建立名字和对象之间的联系，分辨按名访问的对象，并定位该对象。名字空间的管理类似于文件系统的目录树，对象名连同它所处的名字上下文惟一确定一个对象。

事件服务：允许客户对象或服务对象发送消息（事件）到事件通道，由多个接收者接收。

持久服务：提供操作界面持久地在 ODBMS、RDBMS 或简单文件中保存对象。

生命周期服务：定义在软件总线上创建、复制、移动、删除软件构件。

事件服务：又称 OTS（Object Transaction Service），在分布式 CORBA 系统中，支持数据库系统常见的交易概念，即 ACID 特性，支持两阶段提交协议。

外部化服务：允许把一个对象的状态保存在字节流中，及在字节流中恢复并重新构造对象。

CORBAservices 使构件服务者不必考虑系统服务，独立地开发商用对象。然后，系统集成者根据客户的需要，通过多路继承的方式把原有对象和系统服务集成在一起。

CORBAfacilities 为更高级的、直接应用于应用程序的服务提供操作界面。CORBAfacilities 对于开发分布式应用不是必需的，但是在某些情况下是有用的。这些工具提供信息管理、系统管理、任务管理和用户界面。目前正在开发的公共工具包括移动代理、数据交换、工作流、防火墙和商用对象框架等。对象技术的最终目的是提供类似真实世界中实体的中等粒度的软件构件。通过集成生成满足用户需要的应用程序。

ApplicationObjects 主要为某一类应用或一个特定的应用提供服务。它们可以是基本服务、公共支持工具或特定应用服务。

4.1.4 CORBA 的特点

CORBA 规范是由 OMG 提出的一种开放的、分布式对象计算结构，它为异构计算环境的互操作提供了标准。使用 CORBA，能够实现应用程序之间的相互通信，而不论它们的位置、编程语言以及操作系统环境是否相同。CORBA 规范以其自身的优势领导着开放分布处理的发展，它具有如下突出特点：

① 分布计算技术和面向对象编程技术 OOP 相溶合。通过 OOP 的继承性，实现软件代码的重用。CORBA 规范定义的基础是面向对象的设计思想和实现方法，将分布式计算与面向对象的概念相结合也是当前软件系统的普遍模式。

②“代理”概念的引入。代理的基本作用有三个：完成对客户方提出的抽象服务请求的映射；自动发现和寻找服务器；自动设定路由，实现到服务器方的执行。通过代理，用户在编制客户程序时不需了解实现的细节，只需完整地定义和说明所需完成的任务和目标即可。

③ 增加了代理机制后，实现了客户端程序与服务器端程序的完全分离，客户不再同服务器发生直接联系，而仅和代理进行交互。因此在保持调用方式不变的情况下，服务器方和客户方程序都可以自由地修改和升级而无需通知对方。从根本上改变了传统的面向过程调用机制的客户/服务器模式。

④ 提供了“软件总线”的功能。该功能起到类似于计算机硬件总线的作用，只要将应用模块按总线规范做成软插件，插入总线即可实现集成运行，实现了软

构件的即插即用。其中软件总线就是 CORBA 定义的一组独立于语言和环境的接口规范，按照该接口规范开发出来的软件，便可方便地集成到该系统中，而且这个接口规范独立于任何实现语言。

⑤ 设计原则和设计方式的层次化。CORBA 规范仅定义了 ORB 中用到的最基本对象、属性和方法，而面向应用的对象可以在 OMA 的应用对象、领域对象或开发环境中逐层进行定义和实现，CORBA 规范是针对 ORB 通信中间件制定的工业标准，而面向应用的对象定义则可在对象管理体系结构的应用对象或应用开发环境中逐步分层定义和实现。这样“层层扩展”，从而使 ORB 始终处于非常精炼的状态。

CORBA 的这些特点是其充分利用软件技术最新成果的结果，使其成为开放的、基于客户/服务器模式的、面向对象的分布计算的工业标准。

4.1.5　CORBA 的消息处理机制

在新规范出现之前，OMG 提出了 3 种通信模型：

① 双向同步模型，即一般意义上的同步调用机制；

② 单向调用（one-way）模型，即 Client 在调用服务发出请求之后，无需等待而直接返回，又称 fire-and-forget 模型；

③ 延迟同步模型（deferred synchronous），利用延迟查询或分进程查询获取返回结果，可以减少等待。

但是，CORBA 还是缺乏真正意义上的异步消息处理模型，始终无法提高应用系统的并行度，这也极大地限制了 CORBA 本身的发展。因此，在 CORBA3.0 中补充了有关消息处理的规范，它主要包括异步方法调用 AMI（Asynchronous Method Invocation）和独立于时间（或时间无关）的调用 TII（Time Independent Invocation）。

1. AMI 的两种调用模型

AMI 包括两种调用模型：轮询机制（polling）和回调机制（callback）。

①轮询机制。Client 发出调用请求，并立即返回一个 ValueType 类型的 poller。随后 Client 通过 poller 方法对 Server 进行监控，它可以选择使用阻塞或非阻塞的方法获取状态及返回结果。

②回调机制。Client 在调用 Server 的服务时，将 ReplayHander Servant 的引用传送到 Server 端，然后无阻塞地执行下去。当 Server 完成服务时，向 Replay-Hander 发出 Response 请求；然后 Client 便可以通过 ReplayHander 获得服务器返回的结果。因为 Client 无需反复查询是否有结果返回，所以回调机制较轮询机制效率更高。

这两种通信模型的共同特点是客户程序不需要使用附加线程，因此，一个应用程序可以通过单个控制线程来同时管理多个双向操作，达到对一个或多个对象的远程请求处理目的。我们可以使用 polling 模式或 callback 模式来避免 DII 和单向方法的不足。而且，由于 CORBA 消息规范只在客户方增加了异步调用，因此不需要修改服务器方程序，实现和使用都较为方便。

2．TII 工作机理

TII 可以看作是一个特殊的 AMI，但与 AMI 不同的是发出调用请求的对象和接收请求结果的对象可以不同。因此，TII 请求的生命周期可能与发出请求的对象的生命周期不一致。TII 的主要思想是对请求/响应的“存储转发”（store-and forward），这很类似于电子邮件的工作原理。如图 4-6 所示，其中，1、2、3、4：Client 将服务对象 A 的引用经过本地 Router、外部网络 Router 和 Server 端 Router 传至对象 A；5、6、7、8：对象 A 将返回结果利用传输链路上的 Router 反向传至 Client 端的 ReplyHandler。

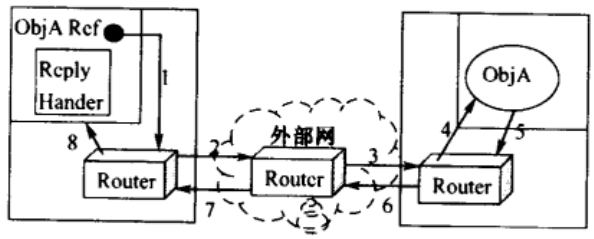


图 4-6 TII 工作机理

当调用请求所需经过的网络环境不好或 Server 暂时不可到达时，它先暂存于所经链路上的节点中，利用 IRP（Interoperable Routing Protocol）为它们提供路由转发服务直至到达 Server 端对象，对返回结果依然。因此 TII 这种工作方式特别适应于网络条件不好或移动的用户。

TII 通常适用于那些需要确保请求能传输到目标对象（在发送请求时，该目标对象可能不与网络相连）的应用程序。CORBA 的 TII 模式与 E-mail 相类似。当用户发送一个 E-mail 后，E-mail 系统存储 E-mail，直到你把计算机与网络相连为止。这时，用户的 E-mail 系统把该 E-mail 发送到 SMTP 服务器上，该 SMTP 服务器再把用户的 E-mail 转发到另一个 SMTP 服务器上，重复上述步骤，直到用户 E-mail 到达接收者为止。

一个 TII 路由器也能接收 CORBA 请求和响应，然后临时存储它们直到它能把这些信息转发到下一条为止，这与 SMTP 服务器接收 E-mail 并进行发送的情况相类似。由于 CORBA 路由器能临时存储 TII 请求和响应，这导致请求和响应

比发送它们的客户程序或服务器程序存活时间长。因此，TII 比较适合于那些偶尔连接网络的客户程序。

3. 消息服务质量

服务质量 QoS（Quality of Service）目前已被人们广泛接受，是指服务性能的聚集效应，它决定用户对特定服务的满意程度，人们通常用它来描述那些用来提高和控制通信资源的实体和技术。许多分布式应用程序对 QoS 有不同的要求，例如某个请求的端延迟、某一节点的最大吞吐量、发送单向消息的可靠性、在客户程序过期前需要等待响应的时间等。OMG 在消息规范中定义了 QoS 构架，从而使得应用程序可以根据需要配置 ORB 和控制 ORB 的行为。该 QoS 构架定义了一组策略对象、管理策略的构架和 GIOP/IIOP（ORB 间通信）策略。

OMG 消息规范中定义的标准 QoS 策略给 CORBA 应用程序提供了更强大的功能和更大的灵活性。下面对标准的 QoS 策略类型作简单介绍：

① 重新绑定（rebind）策略。传统的 CORBA 应用程序没有给客户 ORB 定义标准的方法来处理接收到 LOCATIONFORWARD 的应答。对于是否透明的重新绑定（是传统 CORBA 行为）、或是当连接关闭时重新绑定而不是因为得到 LOCATIONFORWARD 应答而重新绑定、或者不重新绑定等情况，应用程序可以使用重新绑定策略进行有效的选择。而且，消息规范中增加了 CORBA：：Object：：validateconnection 操作，使得应用程序能控制重新绑定而不考虑实际的重新绑定策略。

② 同步策略。该策略允许应用程序能控制单向操作的语义。不同单向调用具有不同级别的可靠性，特征和潜在开销。

③ 请求和响应优先级策略。一个 TII 路由器接收 CORBA 请求和响应信息后，临时存储它们，直到把它们发送到下一个路由器为止。TII 路由器使用的请求策略和响应策略可以决定请求信息和响应信息的存储次序和转发次序。优先级高的请求和响应在路由器队列中得到优先处理。

④ 请求和响应到期策略。应用程序使用这些策略能控制发送请求和响应时与时间有关的方面。例如，一个应用程序能控制 ORB 发送请求或返回响应的时间窗；同样，一个客户方也能控制 ORB 发送请求的相对时间。而且，客户方也能控制从发送请求到取得响应所需的时间。超过了指定时间后，ORB 将触发一个 CORBA：：TIMEOUT 异常。应用程序可以采用适当的方法处理这种异常事件。

⑤ 请求路由策略。应用程序可以指定某个请求不通过路由器而直接用客户 ORB 发送。当然，应用程序也能指定某个请求必须由特定的路由器发送给目标对象。如果某个应用程序选择使用路由器，那么它可以指出是使用异步方法调用

还是采用与时间无关的调用。如果使用路由器来传送请求，则可以使用另外一个策略 Maxhops 来控制到达目标的传送路径上的最大数。

⑥ 排队顺序策略。该策略可以控制通过路由器发送的请求的排队情况，例如，应用程序可以指定排队的顺序、临时顺序，优先级或按照请求的到期时间排队。

4.1.6 CORBA 对象适配策略

1. 对象适配的特点及一般适配策略

服务方 ORB 处理请求的过程如图 4-7 所示。

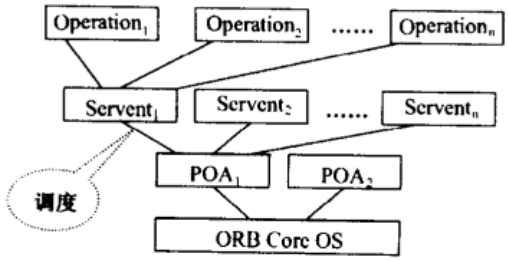


图 4-7 对象适配过程

ORB 核心从操作系统（底层网络传输机制）获得客户方发来的请求消息，对请求进行相关的处理，如对请求头解码等。如果使用了安全服务，则进行相关的审计等。然后 ORB 核心使用 RootPOA 解析对象关键字，提取注册请求对象 POA 的层次结构信息，定位 POA。然后将请求消息送往该 POA，POA 根据所使用的策略来解释对象标识，按照策略所规定的算法来定位实现该请求的服务。

（1）生存期策略（life span policy）

CORBA 中把所有的对象分为暂态（transient）对象和持久（persistent）对象两种。暂态对象的生存期是由创建它的 POA 和服务器进程所决定的，而持久对象的生存期超过了创建它的 POA 和服务器进程。因此暂态对象的状态只是暂时地保存在内存中，并随着服务器进程的消亡而永久地消失；而对于持久对象的状态则是保存在永久存储中（比如硬盘中），即使服务器进程消亡它依然存在。因此管理持久对象的开销比管理暂态对象的开销大得多。二次开发者通过选择 POA 生存期策略值来决定该 POA 管理的是暂态对象还是持久对象。

（2）对象标识符分配策略（ID assignment policy）

当 POA 创建一个对象时，首先它要生成一个在本 POA 中惟一的对象标识（ObjectID），然后将此对象 ID 封装到对象引用中，最终将对象引用传给客户端使用；当客户端向服务端发送请求时，POA 也是根据对象 ID 把请求调度到目标

对象上执行。因此，对象 ID 是 POA 要管理的一个重要实体，如何对它进行管理将直接影响到 POA 调度的效率。CORBA 提供了两种生成对象 ID 的分配方法：系统分配（SYSTEMID）和用户分配（USERID）。二次开发者通过选择对象标识符分配策略值来决定要求 POA 创建的对象 ID 是由系统分配还是由用户分配。如果选择系统分配，则由系统来保证对象 ID 的惟一性；相反，如果选择用户分配，则用户要自己保证其惟一性，如用户给新生的对象 ID 分配一个已存在的对象 ID，那么系统会发出异常给予阻止。

（3）标识符惟一策略（ID uniqueness policy）

在 POA 中存在着一张激活的对象映射表（POA active object map），它保留了所有被激活的对象 ID 和伺服程序的关联。POA 也正是通过这张映射表来调度请求。CORBA 允许两种方案来控制这种关联：惟一映射（uniqueID）和多样映射（multipleID）。二次开发者通过选择标识符惟一策略值来决定采用惟一映射或多样映射。如果选择惟一映射，则同类对象的不同实例不能共用一个伺服程序，对象 ID 与伺服程序之间必须是一一对应；相反，如果选择多样映射，则同类对象的不同实例可以共用一个伺服程序，对象 ID 与伺服程序之间可以是多对一的关系。

（4）隐式激活策略（implicit activation policy）

激活的概念是指把 CORBA 对象和具体化它的伺服程序关联起来，以供 POA 调度请求使用。CORBA 提供了两种激活的方式：隐式激活（implicit activation）和显式激活（no implicit activation）。隐式激活是通过映射语言提供的快捷函数来实现激活操作（如C++ 提供的 this 函数）；显式激活是通过在 CORBA 的 POA 接口中提供的函数来实现激活操作（activate object）。二次开发者通过选择隐式激活策略值来决定采用隐式激活或显式激活。

（5）请求处理策略（request processing policy）

CORBA 提供了 3 种 POA 处理请求的方式：只使用激活的对象映射表方式（useactive object map only）、使用伺服管理器方式（use servant manager）和使用默认伺服程序方式（use default servant）。二次开发者通过选择请求处理策略值来决定使用这 3 种方式中的一种。请求处理策略的选择直接影响到 POA 调度请求的效率和系统对内存的管理。当使用激活的对象映射表方式时，POA 仅通过映射表进行调度；当使用伺服管理器方式时，POA 注册一个伺服管理器对象，如果客户端发出请求的 CORBA 对象还未被激活，这时就由伺服管理器动态地生成一个伺服程序来具体化它；当使用默认伺服程序方式时，POA 注册一个默认的伺服程序，由客户端发送的请求都调度到默认的伺服程序中。

（6）伺服程序保留策略（servant retention policy）

CORBA 提供了两种方式来控制伺服程序的管理：保留方式（retain）和不保

留方式（nonretain）。保留方式是允许 POA 保留激活的对象映射表来调度请求，同时也是允许已经激活的对象在内存中保留着它的伺服程序；不保留方式是 POA 中不存在激活的对象映射表，对已经激活过的对象不在内存中保存它的伺服程序。二次开发者可以通过选择伺服程序保留策略值来控制伺服程序在内存中的分配情况。

（7）线程策略（thread policy）

CORBA 给 POA 提供了两种线程模型：ORB 受控模型（ORB ctrol model）和单线程模型（single threas model）。使用 ORB 受控模型，允许 POA 同时接收和处理多个来自服务端 ORB 的请求；使用单线程模型，则 POA 串行化地接收和处理来自服务端 ORB 的请求。二次开发者通过选择线程策略值来决定使用 ORB 受控模型或单线程模型。

POA 的策略的组合是多种多样、错综复杂的。一方面，7 个策略之间有相互制约的关系，一些策略的选择将限制了其他策略的选择；另一方面，策略的选择也限制了一些函数的使用，如创建对象引用函数（create reference）要求伺服程序保留策略的值为保留方式，否则将产生策略错误（wrong policy）的系统异常。因此策略选择的好坏将直接影响到基于 CORBA 的分布式应用程序性能的优劣。

下面对对象标识符分配策略作进一步的说明。

POA 的对象标识分配策略使用的 SYSTEMID 由对应的对象适配器生成，并负责解释其语义，POA 可在对象被激活时，将对象的位置信息包含在 POA 为该对象生成的对象标识中。包含该对象标识的对象关键字被编码在 IOR 中，服务方通过 IOR 向客户公布所访问对象的位置。客户在请求对象时，并不解释对象关键字，只是将其按照 IIOP 协议编码到请求消息头中。服务方 ORB 接收请求后，将请求转发给对应的对象适配器，对象适配器使用对象标识中的位置信息直接定位实现对象请求的服务，完成对象适配的过程。为了高效利用对象适配器中的资源，应允许位置信息的重用，即当某一对象被取活后，可使用相同的位置信息激活另一个对象，位置信息的具体形式由活动对象集的管理结构决定。

从上述过程的分析可知，对象适配的过程实质就是 POA 通过对对象标识的解释，从所管理的活动对象集中获取实现对象请求服务的过程。对象标识的语义和解释与对象适配策略有关。目前，已成熟的查找策略有：

① 二分法查找策略（binary search）。这种策略算法简单，其关键字表的大小等于搜索集中元素个数，具有最佳的空间效率，其平均比较次数的量级为 O(log2n)，但该策略在用于动态搜索空间时，需要对搜索空间中的元素进行重新排序，因此会影响系统的效率。

② 动态散列策略（dynamic hash）。动态散列在最好的情况下，其比较次数的量级为 O(1)；在最坏的情况下，其比较次数量级为 O(n)。该策略适用于动态搜

索集。

③ 最佳散列策略（perfect hash）。最佳散列函数是静态搜索集在时间和空间效率上的最佳实现。它具有最佳性，即在 O(1)的时间里定位静态搜索集中的关键字；具有最小性，即分配存储关键字的表大小正好与静态搜索集中关键字的个数相等。

④ 线性查找策略（linear search）。该策略适用于静态搜索集和动态搜索集，但该策略较为耗时，需要比较的次数量级为 O(n)。

CORBA 标准允许用户动态地激活和去活对象，因此对象适配器所管理的对象集是一个动态搜索集。上述的策略只有动态散列策略和线性查找策略适用于动态搜索集，这两种策略都通过对对象标识的计算和比较来定位对应的服务，较为低效，尤其是线性查找策略不利于大规模应用系统的开发和集成。基于上述的分析，可以设计一种更为积极、主动的适配策略——直接适配策略（direct dispatching）。

2. 直接适配策略

（1）支持该策略的对象标识

根据以上适配策略的基本机理可知，一般的对象适配策略是通过对对象标识的计算和比较来定位对应的服务，无法避免耗时的比较操作和计算，而且对象标识中必须包含对应对象的位置信息。但是，只包含位置信息的对象标识会使对象适配器因 CORBA 分布计算环境允许对象动态地激活和去活，而对不同的对象生成相同的对象标识，使系统错误地定位相关服务。例如，当服务方在对象适配器 POA1 中激活与服务 a（其类型为 A）相关的对象时，其位置信息为 location，客户方所得到访问 a 的对象引用 aRef 中含有 location。由于应用需求，应用将与服务 a 相关的对象去活，但对象引用 aRef 未被删除。然后，服务方在 POA1 中激活与类型为 B 的服务 b（a 和 b 是不同的服务）相关的对象，POA1 恰好在 location 处激活了与 b 相关的对象，生成了包含 location 的对象引用 bRef。此时，aRef 和 bRef 都是能访问服务 b 的对象引用。当客户通过 aRef 来请求对象时，将会错误地定位服务 b，但 b 并不是完成该对象请求的服务，从而导致系统产生无法预知的结果。因此，对象标识中还应包含对于同一位置信息能保证其惟一性的相关信息，通过该信息应能保证对象适配器对服务访问的合法性检查，例如可使用对象激活时的系统时间作为惟一性信息。因此，支持直接适配策略的对象标识应包含两部分内容：对象的位置信息和惟一性标识。结构如图 4-8 所示。

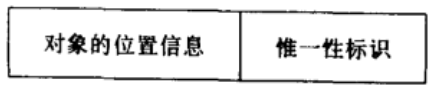


图 4-8　对象标识

(2) 策略的实现机制

基于直接适配策略的基本算法，我们使用动态数组的形式来组织活动对象集，动态数组保证管理空间随着活动对象数的增多而动态增长。为提高对象激活操作的效率，在数组中组织空闲链表，在系统初启时，数组中的所有元素被链接在空闲链表中。数组中的元素定义如下：

```
struct Table-Entry {
    CORBA:: ULong flag-;
    Portable Server:: ServantBase * servant-;
    CORBA:: Boolean is-used-;
    Table-Entry * free-;
}
```

其中，

flag-：该值用于同数组索引一起作为对象标识的惟一性信息和检查对服务访问的合法性；

servant-：该值表示与对象相关的服务，即实现该对象请求的服务；

is-used-：表示数组中对应元素的状态，即该元素是否已被活动对象占用，用 true 表示占用，false 表示空闲；

free-：该值用于指向管理数组空间中的下一个空闲元素。

基于上述数组结构的管理空间，使用数组元素对应的下标作为位置信息，因此对象标识的内容如图 4-9 所示（使用 Table 表示该数组）。对应数组元素中的 flag-按照如下规则改变其值：当系统初启时，flag-被清“0”，当对应元素被占用时，flag-值加“1”后，使用 flag-的当前值和对应元素的下标生成对应的对象标识号。通过对应元素下标与元素的 flag-值结合，能够保证所生成对象标识的惟一性，并且该对象标识只有 8 个字节。

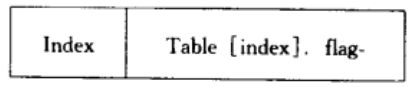


图 4-9 基于数组管理结构的对象标识

基于上述结构，当对象适配器激活对象时，该机制首先从空闲链表中取出第一个元素，将其置为占用状态（is-used- = true），然后对应 flag-加“1”；调整空闲链表，将与该激活对象相关的服务赋值给 servant-，使用 flag-的当前值和对应元素的下标生成对应的对象标识号。当客户通过对象引用请求对象时，对象适配器在对象适配之前，将首先验证对象标识号类型的合法性（检查是否为 8 个字节）。合格后，对象适配器解释对象标识号，分离出数组下标 index 值和对应的 flag- 值。在验证下标 index 的合法性后，通过逻辑表达式：

(Table [index].is-used-) && (Table [index].flag- = = flag-)

来验证请求的合法性（Table 仍表示管理数组）。当上述表达式的值为 true 时，请求合法，返回 Table [index].servant-；否则返回空，由对象适配器根据相关的策略进行处理，完成对象适配的过程。当对象被去活时，通过对象标识进行上述的合法性验证后，将 Table [index].is-used-置为 false，Table [index].servant-置为空，然后将该元素加入空闲链表尾。对于不合法的对象标识返回相应的异常。

通过上述逻辑表达式按照上述算法完全可验证请求的合法性：表达式 Table [index].is-used- 防止了对去活对象的请求；Table [index].flag- = = flag- 则保证了不会产生上述的错误定位服务问题；同时合法性验证也避免了去活对象时产生类似的错误。

直接适配策略是高效的，由于它直接使用对象的位置信息定位实现该对象请求的服务，从而避免了耗时的计算和比较操作，所以它能在 O(1)的时间里完成对象定位。

该策略符合 CORBA 标准，不会影响 ORB 之间的互操作，相关的实现机制简洁、有效。如果在 POA 环境中通过对象适配构件框架，可以隐藏内部适配策略，对 POA 透明，有利于新适配策略的加入和 POA 不同策略的实现。

4.1.7　CORBA 互操作模型

图 4-10 是 OMG 提出的不同对象系统之间互操作的一般模型。图中 B 系统中的目标对象实现通过一个“桥”变换为 A 系统中的一个对象，该对象实际上是 B 系统的目标对象在 A 系统中的视图。这样，A 系统程序实体通过对 A 系统视图对象的访问达到透明访问 B 系统对象的目标。图左边是 A 系统的一个客户，它要将访问请求发送给右边 B 系统的一个目标对象。实现的策略是采用一个桥机制来提供不同对象系统的映射，从而将一方的客户请求透明地传送给另一方。

这里视图所暴露的接口（视图接口）与 B 系统的目标接口是同形的。视图接口负责将 A 系统中的客户请求转换为 B 系统中的目标接口请求。视图是桥机制的组成部分，一个桥可能包括若干个接口。

桥负责不同对象系统之间接口及标准形式的映射。它内置着对 B 系统目标的引用，提供 A 系统和 B 系统的聚集点，同时采用各种措施（如 IPC、RPC、网络和共享存储器等）保持两个系统之间的通信，并保留所有对象之间的语义。

对客户来说，视图就好像 A 系统中真实的对象，调用请求首先以 A 系统的请求形式产生，并转换成 B 系统对象的形式，然后交付给目标对象。结果是 A 系统产生的请求透明地传送给 B 系统的对象实例。

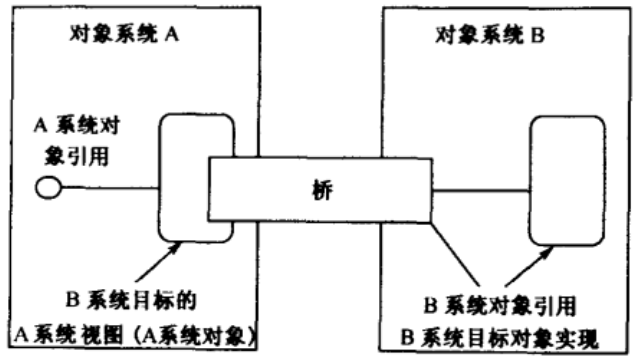


图 4-10 B/A 互操作模型

4.1.8 CCM

CORBA3.0 的最大变化是提出了自己的构件模型 CCM，它的大致结构如图 4-11 所示。

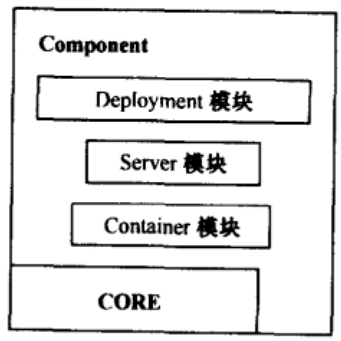


图 4-11 CCM 结构

Deployment 模块：用于部署和配置 Component。

Server 模块：Component 所有最终提供的服务由这个部分组成。

Container 模块：用于实现 Server 所运行的环境，包括安全、事务、均衡负载等功能。通常这一部分放在 POA 中实现。

CCM 不仅最大限度地推行了软件复用，而且也为基于 CORBA 的应用的动态配置提供了很高的灵活性。支持完整的 CCM 需要如下的几个部分。

(1) 提供更加完备的 IDL 与具体语言的映射

OMG 除了定义了 IDL 向 C、C++、Ada、Java、Smalltalk 等语言的映射之外，又增加了 Java 到 Java 的映射。OMG 提出这个规范主要鉴于以下的考虑：目前基于 Java 的分布式应用有两种方式：Java RMI 和 CORBA IDL。相对于后者，

前者易用性好，而且在实现分布式应用时不用再学习另一种语言。但是 Java RMI 缺乏与其他语言的互操作性以及标准的通信协议的支持。而 Java to IDL 规范正好解决了这样一个问题，易于 IDL/IIOP 的映射，实现了语言、协议的互操作性。这个规范的重大意义在于：随着 EJB 的不断完善、发展和应用的不断普及，基于 Java 的分布式应用越来越多，为了能使 CORBA 环境下的 Client 也能使用 EJB 的 Server，可以使用 Java to IDL 对已有的 Java 应用实施逆向工程 IDL，将其改为 CORBA 可用的服务。

（2）支持面向对象设计的脚本语言（Script）

Script 如胶水，将许多对象粘合在一起，形成一个具有完整独立功能的应用服务器。它易学易用，并能够准确无误地反映构件的内部组织（如对象属性、可用方法及其调用机制、异常处理等），而且很容易对系统进行变更和升级。根据 Script 的语法规范，它可以支持 AppScript、VBScript、CORBAScript、Perl 等。

（3）能够实现 CORBA 对象

即可以完成 Client 端的实现，又可以写 Server 端的代码。用 Script 实现的 Client/Server 与其他语义实现的对象没有本质上的区别。但 OMG 认为 CCM 应主要着眼于 Server 对象的能力，这一点与 EJB 是一样的。因此 CORBA 构件的能力主要体现在：利用 CIDL（Component Implement Definition Language）描述每个构件及其关系；利用 XML（eXtensible Markup Language）的 OSD（Open Soft Description）作为程序包的描述子，用于描述有关构件的创建、集成及部署所需要的信息；引入容器编程模型，高度抽象并简化 BOA 和 CORBA 服务；支持独立与构件具体实现的事务控制；此外，CORBA 构件还支持异常的本地处理和传递；对脚本语言不支持的 IDL 类型进行封装和解包以及针对伪对象（pseudo object）提供反射机制。

4.1.9　CORBA 分布式面向对象的分析设计与实现

1. CORBA 分布式分析需要考虑的问题

CORBA 的最大的特点是面向对象，CORBA 系统中最根本的构件部分就是对象及其操作（方法），所以，采用传统的面向对象的分析方法对 CORBA 系统进行系统分析是很自然的。但与传统的分析方法不同的是 CORBA 系统是开放的分布式系统，因此系统分析有其独特之处。

当设计一个分布式 CORBA 系统时，首先要建立起系统的框架模型，形成 IDL 接口文件，也就是从具体的用户需求中抽象出所有的对象、操作及其相互之间的关系，并用 IDL 语言描述出来。由于 CORBA 对用户屏蔽底层通信，因此在设计 CORBA 分布式系统时可以将重点放在系统的功能完成上。建立系统的框架

模型，可以充分体现 CORBA 面向对象的特点，同时体现了对象系统的开放性。

CORBA 分布式对象系统带来的最大问题是客户端和服务器大量请求的存在。在网络环境中，过多的请求将增加网络负担，降低整个系统的性能。因此，如何管理这些请求，充分利用系统资源是为 CORBA 系统建模时必须考虑的一个重要问题。CORBA 请求交互模式分为如下 3 类，可以根据具体需要进行选取。

① 传统的 C/S 模式。在这种模式下，客户对象仅仅是提出请求，而服务器对象仅仅是完成请求的操作，客户和服务器之间通过 ORB 直接进行通信。在这种模式下，客户端仅仅是应用的控制部分，不仅完成用户 GUI 的控制，而且要向服务器传送请求，因此相对比较复杂。而服务器因为完成的操作大都已经确定，相对比较简单。这种模式比较适应于控制部分和被控制部分比较明确的情况。

② 对等交互模式。在这种模式下，客户端和服务器是一种合作的关系，并没有严格的客户端和服务器端的区别。所有的应用相互独立，各自完成一定的操作，同时又可以向其他应用提出请求，因此，相对前一种模式而言，这种对等交换模式比较复杂。

③ 代理交换模式。在这种模式下，客户端和服务器之间存在着一种特殊的服务器：代理（broker）。它代表客户端向服务器提出请求，或代表服务器向客户端提供一定的操作。这里的代理不同于 CORBA 中的 ORB，ORB 仅仅是一个 CORBA 对象，而这里所说的代理是用户在系统建模时提出的作为特殊服务器的应用对象。ORB 只是请求传送的中介，而代理则是服务完成过程中的中介。尽管 ORB 也有中介的功能，但它无法完成负载均衡或选择服务器的功能。图 4-12 给出了 ORB 和代理在 CORBA 系统中的位置关系。

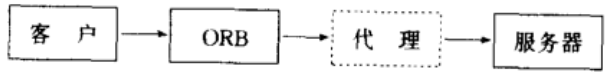


图 4-12 CORBA 中的代理交互模式

代理交互模式用于多个协同工作的服务器环境，或者需要客户端和服务器高度分离的情况。例如，一个打印服务代理服务器，它可以根据用户不同的请求将请求发送型号或类型不同的打印服务器。

一个 CORBA 分布式系统中的交互模式并不固定，应根据环境选择合适的交互模式，以减少系统网络上请求的次数，充分利用网络带宽，提高系统性能。

此外，在设计一个 CORBA 分布式对象系统框架模型时，还应考虑如下几个方面的问题：

① 框架中各对象之间的通信协议问题。尽管 ORB 在应用层屏蔽了底层通信，但必须保证下层通信协议可以为 ORB 所采用。

② 框架中客户端和服务器端之间采用何种通信机制。CORBA 提供 3 种通信机制：同步、异步和单向，3 种通信机制各有其适应的范围和环境。

③ 整个系统运行的软硬件平台。因为各平台之间并不相互兼容，而统一运行环境又是不切实际和比较昂贵的。

④ 如何最少的使用系统资源，比如通信信道、链路等，以充分提高系统性能。

⑤ 其他一些异常情况，比如运行时间无法访问服务器等。这些问题在传统的面向分析中不存在，这与 CORBA 系统的分布特点有关。

2．CORBA 分布式分析的方法和步骤

将面向对象的系统建模应用于 CORBA 系统中，得到以下几个必要的步骤，如图 4-13 所示。

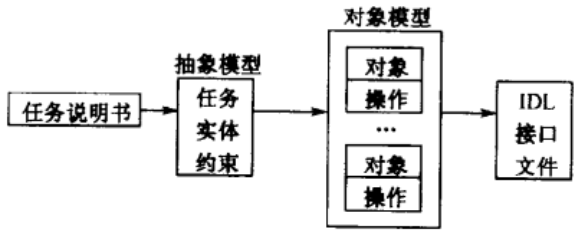


图 4-13　CORBA 系统分析的过程

① 明确系统需求，产生系统的任务说明书。主要明确以下几个方面的问题：系统要完成的任务以及要解决的问题；各个任务执行的实体和环境；数据的安全性；系统中需要进行扩展性的部分。

② 应用任务说明书建立系统的抽象模型。抽象模型中包括根据任务说明书提取出的任务、实体以及相互之间的约束。建立抽象模型有助于整个软件系统的实现、维护、扩充以及重用。

③ 在抽象模型的基础上建立系统的对象模型。将抽象模型的任务作为操作，实体作为对象，然后将操作和相应的对象联系起来，在其上加以约束条件，并对对象和操作加以分组就可以得出 CORBA 分布式系统的对象模型。这样得到的模型是原始对象模型，必须对其进行进一步的加工和改造，以形成最终简单实用的对象模型。

④ 用 OMG IDL 语言描述对象模型，得出 IDL 接口文件，从而形成 CORBA 系统的框架模型。在得到了整个系统的框架模型——IDL 接口文件之后，根据 CORBA 系统的实现结构，可以方便地编译实现客户端存根（Stub）和服务框架（Skeleton），以及扩充接口库，最终实现整个 CORBA 分布式对象应用系统。

3. CORBA 应用开发和实现的主要步骤

基于 CORBA 的分布式应用系统开发的主要步骤为：

① 编写 OMG IDL 接口规范说明文件。该文件描述了服务对象所支持的操作和类型，也就定义了可向该对象提送的请求。对象的接口由接口定义语言 IDL 描述。IDL 的一个重要的特点是其语言独立性。目前 OMG 已定义了 OMG IDL 到多种设计语言的映射。

② 编译 IDL 规格说明文件，生成客户端存根和服务器方框架。由于 IDL 是一种描述性语言，不是程序编写语言，所以使得接口定义和对象实现分开，允许对象用不同的设计语言实现但仍然保持相互之间通信。因此，在编译 IDL 文件时生成的存根和框架可以不是同一种程序设计语言。如开发者在客户端想用 Java 语言开发而服务对象用C++ 语言实现，则可用 IDL2Java 编译成 Java 语言的客户方存根，用 IDL2 C++ 编译成C++ 语言的服务方框架等。

③ 编写对象实现程序。在这里，开发者要将 IDL 文件中定义好的对象所能提供的方法进行具体实现，即须开发者真正花气力书写代码。

④ 编写服务方和客户方主程序结构。服务方主程序首先创建 ORB 和 BOA 对象，然后创建实现对象，最后进入消息分配循环，等待客户方来的请求，并将请求分发给指定的对象。

客户方主程序首先创建 ORB 对象，然后获得服务对象的对象引用，最后通过该对象引用向服务方对象发送请求，请求服务方完成指定的功能。若服务方是基于 Web 和浏览器方式的，则相应地编写 Java Applet 小程序代码。

⑤ 将对象实现程序、服务方主程序和生成的服务方框架编译连接成服务方可执行程序；

⑥ 将客户方主程序和生成的客户方存根编译连接成客户方可执行程序。若服务方是基于 Web 和浏览器方式的，则相应地编写 HTML 超文本文件，链入编写好的客户方 Java Applet 小程序。

⑦ 分别运行服务器方和客户方程序。

4.2 EJB 模型

分布式对象技术特别适合构造灵活的客户机/服务器系统。由于数据和处理逻辑被封装成对象，这就使它们能分布在系统中的任何位置，从而增强分布的颗粒度，使各层相对独立。对象是具有自我管理能力的节点，可以修改它们，而不影响系统中和它交互的其他构件。但是，仅有分布式对象是无法完成以上要求的，还需将它们打包成构件，才能使它们在一起工作。采用分布式对象技术，你

可以通过组装已有的对象来生成新的对象，并用它们来构造符合客户需要的灵活的分布式应用程序。

4.2.1　JavaBean、EJB 和 RMI 概述

1．JavaBean

1996 年 10 月 SUN 公司推出的 JavaBean，经过几次升级已成为比较重要的软件构件，按照 SUN 公司的定义，JavaBean 是一种“能在开发工具中被可视化操作的、可重用的软件构件”。它的最初目的是定义一种 Java 的软件构件模型，使第三厂方可以生成和销售供其他人员使用的 Java 构件。Bean 可以被放置在“容器”中，提供具体的操作性能。

JavaBean 有如下的特点：

① 支持内省（introspection），使构件可以发表其操作和属性；

② 支持客户化配置（customization），使开发人员可以控制每个构件的外观和行为；

③ 支持事件（events），是一种连接 Bean 的简单通信机制；

④ 支持属性（properties），对构件布置的控制，包括它所占有的实际空间和在容器中相对其他构件的放置关系；

⑤ 支持永久性（persistence），使构件能够存放当前的状态，并在以后恢复。

并非所有的 Java 类都能包装成 JavaBean，JavaBean 只适应于那些能被可视化操作和配置的软件模块。一些为编程提供功能的函数的类通常放在类库中。

Bean 大致分为两类：简单 Bean 和合成 Bean。

简单 Bean 由可见 Bean 和不可见 Bean 组成。前者必须是 java.awt.Component 类的子类，它们构成了应用程序的 GUI 界面；后者可以是任意的 Java 类，用于包装数据的处理逻辑。

合成 Bean 则包含一组可见和不可见的 Bean，它必须是下面 3 个 Java 类的子类：

① Applet 类，这种类主要构造 Web 应用；

② Frame 类，构造 GUI 的合成 Bean；

③ Panel 类，构造可重用的 GUI 元素，可以嵌入到其他 GUI 元素中或用来组成其他 Bean。

构件是可识别的并定义好边界的软件片，构件模型定义构件的外形和边界。包容器可以是一个构件，为它包容的所有构件提供运行的上下文。JavaBean 规范描述了 Java 的构件系统结构。JavaBean 构件模型以 Java 类为基础，并规定程序员须遵循的使 JavaBean 可重用及用可视工具管理的规则。Bean 既能在容器中运

行，也能在工具程序、应用、Applets 和 HTML 页中运行。JDK 本身是运行 JavaBean 的框架。Bean 和 applet 不同，Applet 可以是一个 Bean。

Bean 可以将其状态保存到文件并用 JAR（Java ARchive）打包。借助相应的 Beaninfo 类可以内省 Bean。因此，其他的 Bean 或工具能够动态发现一个 Bean 的有关信息。一个 Bean 提供 Customizer 类，允许在设计时用工具定制它的行为，通常通过属性编辑器设置 Bean 的属性。开发人员的工作是：从头开始创建新的 Bean，或者通过定制已经存在的 Bean，靠事件将它们连接在一起，装配出新的应用。这种软件生产线的概念是现代软件开发模式的方向。

在 CORBA 系统中，IDL 定义的对象是工作和分布的单位，CORBA 对象为跨语言和跨网络的互操作提供标准界面。因此，根据定义，CORBA 对象是构件。JavaBean 使用可移植的构件下层结构扩展 CORBA，CORBA 则用分布式的、支持多语言的构件下层结构扩展 JavaBean。

2．EJB

EJB 规范是 SUN 公司于 1997 年 12 月发布的 JavaBean 构件模型。一个 EJB 是特定的在服务器上运行的 JavaBean，并且 EJB 能在可视化的工具下装配成新的应用。服务器构件是运行在服务器上的被包容器管理的 Bean。包容器充当构件协调者的角色，使用标准的 JAR 从其包容器或工具中导入 EJB。一旦 EJB 处于包容器内，就可以加入事件处理、状态管理、自动激活或撤消等功能。EJB 框架的优点是以说明的方式定义服务器方运行时的属性。因此，可以通过可视工具管理和设置服务器方构件的属性，包括它们的事务处理、安全性和状态管理器等。

EJB 规范是位于服务器方的 JavaBean 和一种新的构件协调者 OTM（Object Transaction Monitor，EJB 称之为包容器）之间的协定，OTM 可看成是建立在 ORB 之上的 TP Monitor，提供运行服务方构件的框架，它负责激活构件或撤消构件，协调分布式事务，通知构件事件的到来，以及自动管理持久构件的状态。CORBA 没有定义服务器方构件协调者的框架，处理事务、构件打包、自动状态管理等工作留给开发商来完成。另外，CORBA 缺少一个完整的服务器方构件模型。过去 ORB 开发商的重点是 CORBA 的互操作性，但没有服务方构件的概念，无法用构件表示被管理的对象（即使用可视工具可操作的、集成各种 CORBA 服务的对象）。CORBA Bean 不是被管理的服务器方对象，只是简单地使用常规的 CORBA 对象利用可视工具管理，可能导致正在被 ORB 开发商定义的服务器方构件无法互相交换。这正是 EBJ 解决的问题。EBJ 定义服务器方构件模型和一个构件协调者框架，并与 CORBA 兼容。这样，就具备了基于 JavaBean 的客户方和服务器方的构件模型。EJB 还需要在服务器方支持除 Java 外的其他语言，方能成

为 CORBA 标准。Web、JavaBean、CORBA 结合，产生了新一代分布式、基于构件的对象 Web。

3. RMI

RMI（Remote Method Invocation）是 JDK1.1 引入的 ORB，可看作是 Java 的 RPC 机制，它定义了一组远程接口，可用于生成远程对象，客户机可以像调用本地方法一样调用远程对象。RMI 直接把分布式对象模型嵌入到 Java 语言内部，使 Java 程序员自然地编写分布式程序，不必离开 Java 环境或者涉及 CORBA IDL、以及 Java 到 CORBA 的类型转换。然而，RMI 不遵循 CORBA 标准，基本上是 Java to Java 技术，它要求客户方和服务器方程序由 Java 编写，难以实现与其他语言编写的对象间的互操作。

RMI 包括 3 层：Stub/Skeleton、远程引用层和传输层，如图 4-14 所示。

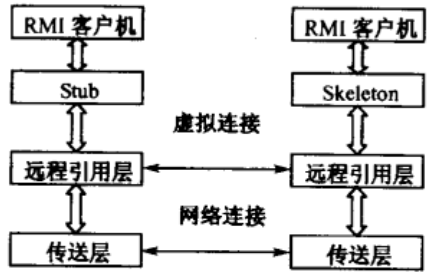


图 4-14　RMI 结构

当客户机调用远程方法时，调用请求首先传送到客户机的 Stub。客户机用 Stub 作为远程对象的代表，然后，远程引用层将调用请求传递到服务器对象的 Skeleton，由 Skeleton 进行服务器方法的调用。

RMI 可以用 JNMI（Java Native Method Interface）与现有的系统集成。同样，它也可以用 JDBC 与关系数据库连接。这样，可以用 RMI 与已有的非 Java 实现的服务器进行通信。

传统的 RPC 系统，如 DCOM 或 CORBA，只能传递数据作为参数，而 RMI 可以传递各种类型的参数，包括对象和实现。这使得对象的分布性得到了进一步的提高。

4.2.2　EJB 的体系结构

EJB 分布式应用程序是基于对象组件模型的，底层的事务服务用了 API 技术。EJB 技术简化了用 Java 语言编写的企业应用系统的开发和配置。EJB 技术定义了一组可重用的组件。你可以利用这些组件，像搭积木一样建立你的分布式应

用程序。当你把代码写好之后，这些组件就被组合到特定的文件中去。每个文件有一个或多个企业 Bean 和一些配置参数。最后，这些企业 Bean 被配置到一个装了 EJB 容器的平台上。客户能够通过这些 Bean 的 home 接口，定位到某个 Bean，并产生这个 Bean 的一个实例。这样，客户就能够调用 Bean 的方法和远程接口。

EJB 服务器作为容器和底层平台的桥梁管理着 EJB 容器和函数。它向 EJB 容器提供了访问系统服务的能力。所有的 EJB 实例都运行在 EJB 容器中。容器提供了系统级的服务，控制了 EJB 的生命周期。EJB 中的一些易于使用的管理工具有：

① 安全配置描述器（security deployment descriptor）。安全配置描述器定义了客户能够访问的不同的应用函数。容器通过许可授权的客户访问这些函数来达到客户的目的。

② 远程连接（remote connectivity）。容器为远程连接管理着底层的通信机构，而且对企业 Bean 的开发者和客户都隐藏了通信细节。EJB 的开发者在编写应用方法的时候，就像是在调用本地的平台一样。

③ 生命周期管理（life cycle management）。客户简单地创建一个企业 Bean 的实例，而容器管理着企业 Bean 的实例，使企业 Bean 能实现最大的效能和内存利用率。容器也能够使企业 Bean 失效，保持众多客户共享实例池。

④ 事务管理（trasction management）。配置描述器定义了企业 Bean 事务处理的需求。容器管理着那些管理分布式事务处理的复杂机构。这些事务可能要在不同的平台之间更新数据库。容器使这些事务之间互相独立，互不干扰。保证所有的更新数据库都是成功发生的，否者，就回滚到事务处理之前的状态。

4.2.3 EJB 中的角色

一个完整的基于 EJB 的分布式计算结构由 6 个角色组成，这 6 个角色可以由不同的开发商提供，每个角色所做的工作必须遵循 SUN 公司提供的 EJB 规范，以保证彼此之间的兼容性。这 6 个角色分别是 EJB 组件开发者（enterprise bean provider）、应用组合者（application assembler）、部署者（deployer）、EJB 服务器提供者（EJB server provider）、EJB 容器提供者（EJB container provider）和系统管理员（system administrator）。

（1）EJB 组件开发者

EJB 组件开发者负责开发执行商业逻辑规则的 EJB 组件，开发出的 EJB 组件打包成 jar 文件。EJB 组件开发者负责定义 EJB 的 remote 和 home 接口，编写执行商业逻辑的 EJB 类，提供部署 EJB 的部署描述文件（deployment descriptor）。部署描述文件包含 EJB 的名字、EJB 用到的资源配置，如 JDBC 等。EJB 组件开发者是典型的商业应用开发领域专家。

EJB 组件开发者不需要精通系统级的编程，因此，不需要知道一些系统级的处理细节，如事务、同步、安全、分布式计算等。

(2) 应用组合者

应用组合者负责利用各种 EJB 组合一个完整的应用系统。应用组合者有时需要提供一些相关的程序，如在一个电子商务系统里，应用组合者需要提供 JSP (Java Server Page) 程序。

应用组合者必须掌握所用的 EJB 的 home 和 remote 接口，但不需要知道这些接口的实现。

(3) 部署者

部署者负责将 EJB 的 jar 文件部署到用户的系统环境中。系统环境包含某种 EJB 的服务器和 EJB 容器。部署者必须保证所有由 EJB 组件开发者在部署文件中声明的资源可用。

部署过程分两步，首先，部署者利用 EJB 容器提供的工具生成一些类和接口，使 EJB 容器能够利用这些类和接口在运行状态管理 EJB。部署者安装 EJB 组件和其他在上一步生成的类到 EJB 容器中。部署者是某个 EJB 运行环境的专家。

某些情况下，部署者在部署时还需要了解 EJB 包含的业务方法，以便在部署完成后，写一些简单的程序测试。

(4) EJB 服务器提供者

EJB 服务器提供者是系统领域的专家，精通分布式交易管理、分布式对象管理及其他系统级的服务。EJB 服务器提供者一般由操作系统开发商、中间件开发商或数据库开发商提供。

在目前的 EJB 规范中，假定 EJB 服务器提供者和 EJB 容器提供者来自同一个开发商，所以，没有定义 EJB 服务器提供者和 EJB 容器提供者之间的接口标准。

(5) EJB 容器提供者

EJB 容器提供者提供 EJB 部署工具，为部署好的 EJB 组件提供运行环境。EJB 容器负责为 EJB 提供交易管理和安全管理等服务。

EJB 容器提供者必须是系统级的编程专家，还要具备一些应用领域的经验。EJB 容器提供者的工作主要集中在开发一个可伸缩的、具有交易管理功能的、集成在 EJB 服务器中的容器。EJB 容器提供者为 EJB 组件开发者提供了一组标准的、易用的 API 访问 EJB 容器，使 EJB 组件开发者不需要了解 EJB 服务器中的各种技术细节。

EJB 容器提供者负责提供系统监测工具，用来实时监测 EJB 容器和运行在容器中的 EJB 组件状态。

(6) 系统管理员

系统管理员负责为 EJB 服务器和容器提供一个企业级的计算和网络环境，并负责利用 EJB 服务器和容器提供的监测管理工具监测 EJB 组件的运行情况。

4.2.4 EJB 的特点

EJB 是一种基于构件的开发模型，它是 Java 服务器端服务框架的规范。EJB 详细地定义了一个可以方便地部署 Java 构件的服务框架模型，用于创建可伸缩、多层次、跨平台、分布式的应用，并可创建具有动态扩展性的服务器端的应用。EJB 服务器实现了对 EJB 对象构件的管理，并提供对系统服务的访问。EJB 服务器也可以提供其他软件接口访问标准，如优化的数据库访问接口，对 CORBA 服务的访问等。EJB 技术具有以下主要特点：

① EJB 以构件的形式组织服务器。EJB 构件是直接用 Java 语言编写的服务器端构件，Java 语言的跨平台特性使得 EJB 构件可以方便地移植到各种操作系统平台和 EJB 服务器上。

② EJB 构件实现仅需考虑应用需求，其系统级服务诸如事务管理、安全性、构件生命周期和线程等，都是通过 EJB 服务器自动进行管理的。

③ EJB 的体系结构具有面向对象、分布式、跨平台、可扩充性、安全性以及便于开发等优点。同时它还是以协议为中心的，任何协议都可以被利用，如 IIOP、HTTP、DCOM。

由此可见，EJB 技术很好地补充了 CORBA 体系规范，在 CORBA 体系上引入 EJB 技术使得建造应用程序更为容易。事实上最近发布的 CORBA 规范中在建立 Java 语言的 CORBA 构件时已大量参考了 EJB 的体系结构。

4.2.5 利用 EJB 进行开发的步骤

开发 EJB 的主要步骤一般包括开发、配置和组装几个方面。

① 开发。首先要定义三个类：Bean 类本身、Bean 的本地接口类和远程接口类。

② 配置。配置包括产生配置描述器（一般为一个 XML 文件）、声明企业 Bean 的属性、绑定 Bean 的类文件（包括 Stub 文件和 Skeleton 文件）。最后将这些配置都放到一个 jar 文件中。还需要在配置器中定义环境属性。

③ 组装应用程序。包括将企业 Bean 安装到服务器中，测试各层的连接情况。程序组装器将若干个企业 Bean 与其他的组件结合起来，组合成一个完整的应用程序；或者将若干个企业 Bean 组合成一个复杂的企业 Bean。

4.3　COM、DCOM 与 COM+

COM、DCOM 和 COM+ 是微软公司在不同时期的软件构架模型和标准。DCOM 是 COM 适应于 Internet 分布式应用的扩展。仅用 DCOM 很难构架企业级的应用，必须与微软的其他产品相配合。其中在交易管理方面有微软的交易服务器（MTS），它与 CORBA 的 OTS 相似；在消息管理方面，有微软的消息队列服务器（MSMQ）。而这一切又是基于 COM+ 来实现的。

4.3.1　OLE/COM

OLE/COM 可以解决 Windows 环境下不同对象间的复合问题。复合分为连接和嵌入两种方式。对象间有两个层次的复合，一个是图形对象间的复合和激活；另一个是程序一级的对象的集成。在 OLE Automation 中，服务方对象实现客户方发出的服务请求。图 4-15 给出了 OLE 服务器对象实现模型。

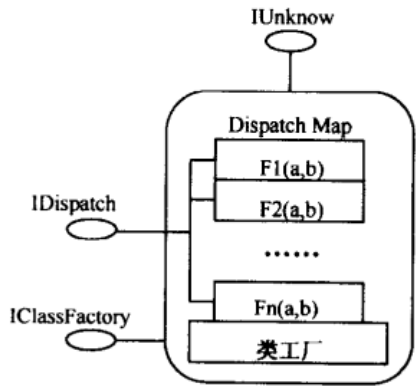


图 4-15　OLE Automation 服务器对象结构

外部程序对 OLE 进行访问主要通过 OLE 服务器对象的接口进行。在 OLE 标准中，接口就是在对象上实现的一组语义相关的功能。在实际中，指向接口的指针就是指向对象实现的指针函数表的指针。

在 OLE Automation 服务器的实现中提供了 3 个基本的接口：IUnknow、IDispatch、IClassFactory。其中 IUnknow 是所有接口的基类，它提供了 3 个基本的操作：QueryIterface、AddRef 和 Release；IDispatch 是向外部展示 OLE Automation 服务器对象实现中的属性和方法的接口；IClassFactory 是创建服务对象的接口。

OLE Automation 服务器的分发映射宏（DISPATHMAP）在对象内部产生一个数据表，IDispatch 的 Invoke 成员函数可以读到该表，客户程序通过 IClassFac-

tory 接口创建一个对象，同时获得了该对象的 IDispatch 接口指针，通过该指针调用 Invoke 成员函数访问服务器；服务器对象会利用分发映射对 Invoke 所提供的参数进行解码，调用相应的对象实现来实现相应的任务。

在 COM 系统中，客户对构件对象功能的调用接口一般采用 COM-IDL 来描述。COM 定义了两类服务器：进程内服务器（inprocess）和进程外服务器（out-process）。

我们熟知，微软对象连接与嵌入技术 OLE 就是建立在 COM 的基础上，使满足这种接口的系统能交互，如 Word 文档中可嵌入 PaintBruch 的图画对象，其中 COM 是 OLE 对象的通信基础设施。在 COM 的底层中部件对象调用关系表示在多张功能表上，每个功能表有一系列指针，指向其界面的功能实现。其中最基本的接口 IUnknown 可通过界面查询功能访问界面的目录。客户在实际调用时相当于访问动态链接库 DLL（Dynamic Linking Libraay）。

进程内服务器即本地机上的 DLL；进程外服务器分为两类，一是本地机上的 EXE 可执行程序，二是远程机上的 DLL 或 EXE 程序。服务器内部包括构件接口的实现（interface implement）和类工厂（class factory），类工厂生产构件对象，将对象的接口指针返回给客户。构件服务器的定位由 COM 库完成并返回对象指针。COM 对象位置的透明性处理由 COM 的服务控制机制保证。进程外的对象必须先调用服务控制机制提供的代理，代理生成服务对象的远程过程调用 RPC。基于 COM 的系统调用原理如图 4-16 所示。

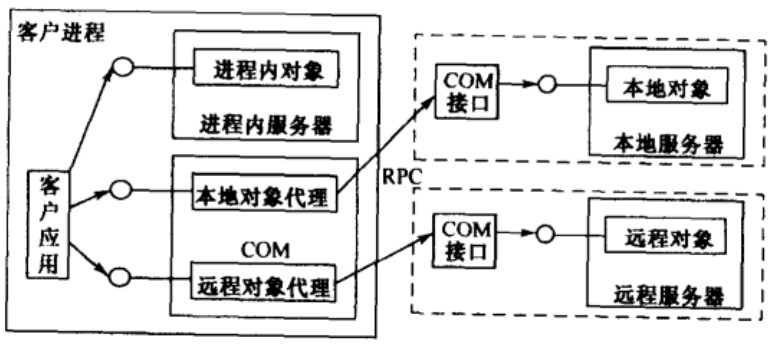


图 4-16 基于 COM 的系统调用原理

4.3.2 COM 的进一步描述

1. COM 对象

COM 定义了创建构件的标准，也定义了构件和它们的客户之间能够交互的方式，这种方式通过 COM 对象实体来进行。客户程序不需要关心构件模块的名

称和位置，但必须知道自己在与哪个 COM 对象进行交互，这就要求对对象进行标识。在 COM 的规范中，每个对象有一个 128 位的全局惟一标识符 GUID (Globally Unique Identifier) 来标识，称为 CLSID (Class Identifier，类标识符或类 ID)。用 CLSID 标识对象可以保证（概率意义上）在全球范围内的惟一性。当系统中含有这类 COM 对象的信息，并包括 COM 对象所在的模块文件以及 COM 对象在代码中的入口点时，客户程序就可以由 CLSID 来创建 COM 对象。

虽然 COM 对象是客户程序进行交互的实体，但 COM 规范对 COM 对象并没有实现方面的要求。它是建立在二进制一级基础上的面向对象构件的对象实体，所以 COM 对象具有封装性和可重用性。COM 对象的数据成员的封装以构件模块为最终边界，对于对象用户是完全透明的、不可见的。COM 对象的可重用性表现在 COM 对象的包容和聚合，而不论哪一种形式，COM 对象的重用都是动态的，一个对象可以完全使用另一个对象的所有功能。

在 Windows 系统平台上，一个 COM 构件或是一个 DLL 文件，或是一个 EXE 文件。一个构件程序可以包含多个 COM 对象，每一个 COM 对象可以实现多个接口。它们的关系可展现在图 4-17 中。

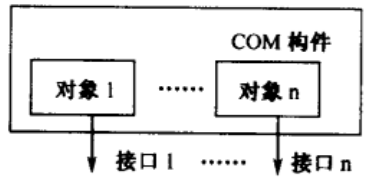


图 4-17　构件、对象和接口的关系

2．COM 的二进制接口

COM 对象是客户程序与构件程序交互的实体，然而客户对某个对象的任何请求都必须且只能通过该对象实现的接口来完成。接口是一组逻辑上相关操作的定义的集合，是客户和对象之间的一种协议，它暗示了对象实现的功能，规定了输入和输出的参数。与 COM 对象的标识一样，每个接口也由一个叫 IID（Interface ID）的 GUID 标识。客户通过 GUID 获得接口指针就可以调用其相应的成员函数，如图 4-18 所示。

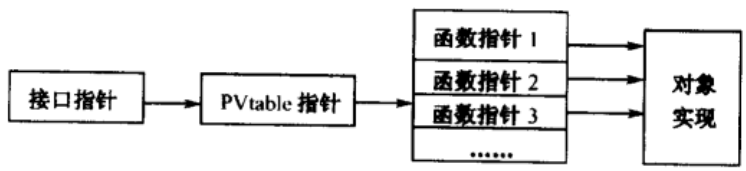


图 4-18　接口结构图

其中 Vtable 为虚函数表，即接口函数表。表中每一项为 4 个字节长的函数指针，每个函数指针与对象的具体实现连接起来。COM 的接口规范并不是建立在任何编程语言的基础上，而是规定了二进制标准，任何语言只要有足够的数据表达能力，就可以对接口进行描述，从而可以用于和构件程序有关的应用开发。COM 的接口具有不变性和继承性，即约定的接口相对稳定及在其接口基础上可进一步扩展。

COM 的接口规范规定，任何 COM 接口都必须从 IUnknown 接口继承而来。IUnknown 接口提供了两个重要的特性：生存控制和接口查询。通过引用计数 AddRef( ) 和 Release( ) 可以有效地控制对象的生存周期，通过接口查询函数 QueryInterface( ) 来完成接口之间的跳转，从而实现对 COM 构件的灵活调用。

3. COM 的客户/服务器模型

COM 对象通过接口来提供服务，实现客户和服务器之间的交互。客户和服务器之间的关系可分为 3 种类型：进程内的、本地的（位于同一主机的不同进程）及远程外的（不同主机，通过网络来通信）。在进程内，服务器构件和客户程序运行在同一地址空间中，一旦客户程序与构件程序建立通信关系，客户程序就可以通过得到的接口指针直接调用服务器对象的成员函数。在同一机器上的不同进程，COM 采用了本地过程调用 LPC（Local Procedure Call）实现进程外构件与客户程序的通信，其过程如图 4-19 所示。对于不同主机的构件程序与客户程序的通信，将在 DCOM 中介绍。

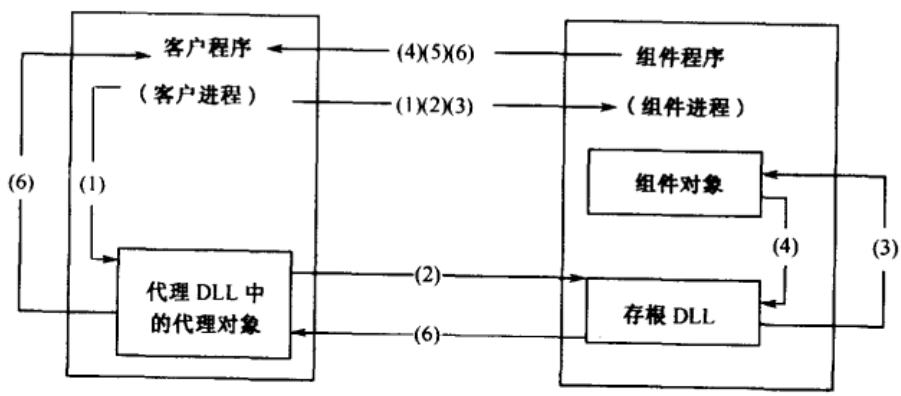


图 4-19 进程外构件与客户程序协作的结构图

(1) 客户调用接口成员函数；　(2) 代理对象通过 LPC 调用组件存根

(3) 调用组件对象成员函数；　(4) 服务完成后返回

(5) 存根 DLL 通过 LPC 返回结果；　(6) 代理对象返回最终结果

4．COM 库

COM 除了定义构件程序和客户程序交互的规范以外，也提供了 COM 的实现部分即 COM 库。COM 库充当了构件程序和客户程序之间的桥梁，尤其是在构件对象的创建过程中，以及在对象管理、内存管理和一些标准化操作等方面起着重要的作用。

在 Microsoft Windows 操作系统环境下，COM 库以 .DLL 文件的形式存在，其中包括以下内容：

① 提供少量的 API 函数，用来加速客户和服务器 COM 应用程序的创建。

② 对于客户，COM 提供基本的对象实例化函数；对于服务器，COM 提供一些对对象访问的支持。

③ COM 通过注册表查找本地服务器，即 EXE 程序、程序名及 CLSID 的转换等。

④ 透明的远程调用，适应于对象在本地或远程服务器上运行的情况。

⑤ 一个标准机制，允许一个应用程序控制内存在其进程中的分配。

5．COM 的主要特点

COM 是一种构造软件构件的二进制标准，而面向对象技术是构件式设计思想的基础。所以，COM 规范所定义的构件模型既具有面向对象的特点，又有其区别于面向对象的特征，主要表现在以下三方面。

(1) 语言无关性

COM 规范的定义不依赖于特定语言，它所采用的是一种二进制代码级的标准，而不是源代码级的标准。所以，编写构件对象所使用的语言与编写客户程序所使用的语言可以不同，只要它们都能生成符合 COM 规范的可执行代码即可。COM 的语言无关性为跨语言的合作开发提供了统一的标准。目前，差不多每种语言在实现时都提供对 COM 的支持，如 C++，Visual C，Visual Basic，Visual J++，Delphi，C++ Builder 等。

(2) 进程的透明性

COM 构件分为进程内构件与进程外构件，但当客户程序创建 COM 对象时可以使用一致的方法，所有实现细节对客户来说都是透明的，这就是创建 COM 对象的进程透明特性。同样地，客户调用 COM 接口也具有进程透明特性。因为客户程序创建进程外的构件对象后，得到构件对象的接口指针，通过该指针，就可以调用构件对象的成员函数。即使这种接口调用是间接进行的，其中也采用了列集（marshaling）和散集（unmarshaling）技术，但客户程序调用接口成员函数就如同调用在进程内的函数一样。

(3) 可重用性

对象重用是 COM 规范很重要的一个方面，它保证 COM 可用于构造大型的软件系统，使复杂的系统简化为一些简单的对象模块，体现了面向对象的思想。COM 重用性是建立在构件对象的行为方式上，它指示了 COM 对象如何重用已有的 COM 对象功能。COM 重用性的实现有两种途径：包容和聚合。

① 包容。假设对象 A 包含了对象 B，当对象 A 需要调用到对象 B 的功能时，它可以简单地把实现交给对象 B 来完成，虽然对象 A 和对象 B 支持同样的接口，但对象 A 在实现接口时实际上调用了对象 B 的实现。

② 聚合。对象 A 只简单地把对象 B 的接口递交给客户即可，对象 A 并没有实现对象 B 的接口，但它把对象 B 的接口也暴露给客户程序，而客户程序并不知道内部对象 B 的存在。

COM 除了以上 3 个主要特性外，还有以下重要的特性。

(4) 支持间接定位技术

即服务器对象对客户的请求不直接作出响应，而是把请求转发给另一个服务器对象，由它去响应请求。利用间接定位，可以解决系统的负载平衡和容错性。

(5) 提供安全性机制框架

在 Windows 平台上，COM 提供两种类型的安全性，即激活安全性和调用安全性。前者指 COM 对象如何被安全地启动，客户如何与对象建立连接以及如何保护公共的资源；后者指在已经建立连接的基础上，客户调用构件程序的安全保护问题。

(6) 多线程特性

可使多个 COM 对象运行在进程的不同线程中，避免多个对象在进程中相互等待，利用系统提供的多任务机制使对象同时运行，从而使应用程序运行得更加流畅。

4.3.3 基于 COM 的构件化程序设计方法

应用构件化程序设计方法（在此针对 COM，但也适应于 COM+）必须遵循一定的标准和规范，目前比较成熟的规范有 COM、CORBA 和 J2EE，它们相互竞争并在一定的范围内取得了极大的成功，已成为许多软件支持的标准。

组件化（或构件化）程序设计方法是对面向对象程序设计方法的进一步抽象，它在层次软件体系结构中扮演着重要的角色。组件化程序设计方法强调真正的软件重用和互操作性，其中组件的产生和装配是程序设计的两大核心。可重用组件库不同于对象库，组件库保存了一些经过测试的组件，并包括组件的细节说明文档。

从组件的产生角度考察，组件的程序设计方法特点如下：

① 编程语言与开发环境的独立性。对于不同的开发人员采用不同的开发语言工具开发的组件，只要符合 COM 标准，即可进行组件的集成，进而形成相应的软件系统。这种开发过程完全与开发环境无关，是一种理想的开发模式。

② 组件位置和组件进程的透明性。这种特性为大规模软件的构架提供了有力的保证。

③ 组件功能的易扩充性。组件对外提供的服务是通过组件的接口进行的，新的服务的增加仅需增加新的接口即可，不会影响以前接口支持下的用户程序。这为组件功能的扩充提供了方便。

④ 组件库的可扩展性。利用 COM 的包容和聚合模型方便地进行组件库的扩展和新组件的加入。

⑤ 具有强有力的基础设施和系统一级的公共服务。基础设施包括组件的检索、组件的远程启动和对组件统一管理的 MTS（COM＋）等，是软件获得重用性、可移植性和互操作性的保障。公共服务包括命名服务、永久对象服务和统一数据传输等。

在基于组件的开发中，可重用组件库扮演着重要的角色。基于组件的开发其实质是基于组件库的开发，这种开发模式大致由 6 个相关联的过程组成（图 4-20）：

① 标识过程。标识一个可重用的组件。

② 验证过程。判定可重用组件所声明的功能特性和性能特性。

③ 分类描述过程。对组件进行使用方法、适应范围和接口等的描述和说明。

④ 检索过程。根据指定的属性找到所需的组件。

⑤ 定制过程。对检索到的组件进行修改、裁剪和配置，以满足用户的需要。

⑥ 组合过程。利用定制好的组件构造应用系统。

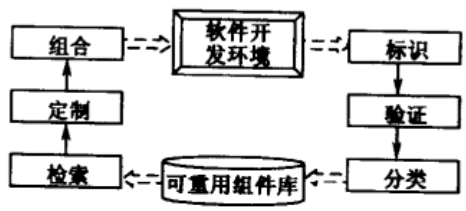


图 4-20　基于组件库的软件开发模式

4.3.4　分布对象构件模型 DCOM

DCOM 可以看作 COM 在分布式和网络环境下的延伸，即在 DCOM 中，各个构件（分布式对象）不再仅仅局限于单机，而是可以扩展到整个网络中。

由于微软产品在全球的成功与巨大影响，因此以服务于微软技术为主要目标

的标准化接口模型 DCOM 也有不可忽视的支持队伍。当然由于 DCOM 完全受微软支配，不是一个开放的标准，又呈现出垄断接口标准的野心，从而受到相当一部分力量的警惕，甚至敌视。

DCOM 已经在 Windows NT 4.0 中发布，但它只适应于 Wintel 平台，即操作系统属于 Microsoft Windows 系列的平台。为了弥补这一不足，微软联合了另一家软件公司——美国的 Software AG 与其合作，并开发了企业版 DCOM (DCOM FTE)，从而将 DCOM 扩展到从 PC 到 Unix 甚至大型主机的企业级范围，这样单个的构件可以在整个企业范围内迁移并完成其功能。DCOM FTE 为企业级的商业计算任务提供了完整的解决方案，而且，DCOM FTE 可以集成原先的非面向对象的应用，原先在大型主机上开发的软件仍旧可以在新的面向对象的环境下继续为客户提供服务，有效地保护了用户的先期投资。

当客户与部件位于不同机器时，DCOM 仅仅用网络协议来代替本地进程之间的通信，参见图 4-21。客户与部件都不会知道连接它们的线路比以前长了许多。在分布应用中，部件的位置对开发人员或用户完全透明。不必改变部件的源码器，不必编译，就可方便地重新配置部件连接方式。

尽管 DCOM 不折不扣地采用了 OO 模型，但其实现远程调用的方法不是消息传递，而是 RPC。DCOM 的大致工作流程如下：

图 4-21 DCOM 不同机器上的 COM 部件

① 客户初始化；
② 服务器激活（客户端）；
③ 服务器激活（服务器端）；
④ 调用类工厂；
⑤ 多查询接口（MQI）；
⑥ Proxy/Stub 装载；
⑦ 方法调用。

第⑥步真正调用了提供服务的对象的方法，第⑤步则完成调用参数的打包，即将其转成 RPC 的规定格式，该操作由客户端的 Proxy/Stub 动态链接库完成。这里的 Proxy 是提供服务的对象（可能在另一台机器上）在本地的映像（shadow object），客户对象把它当作远程的服务提供者对待。

驻留在客户端的 Proxy/Stub 动态链接库是由定义服务对象向外提供接口的

IDL 文件编译来的。在创建 DCOM 服务时，首先要写一段 IDL 代码来定义和描述该服务对象提供的接口，这一点与 CORBA 的 IDL 完全一样。随后 IDL 文件用 MIDL（Microsoft IDL）编译器进行编译后产生用于 Proxy 和 Stub 的 C 源程序，该源程序编译后得到客户对象使用的 Proxy/Stub DLL，也可以得到 Visual Basic 之类的高级工具使用的类型库（type library）。

总之，DCOM 是微软技术在分布式环境中的扩展，它在微软产品体系中显然是很合适的。DCOM 的支持者也在从事着在 Unix 系统中发展 DCOM 的工作，以期达到异构平台上的互操作性。

4.3.5　COM +

1．COM + 简介

COM + 倡导一种新的设计概念，把 COM 构件提升到应用层，把底层细节留给操作系统，使 COM + 与操作系统的结合更加紧密。COM + 的底层结构仍然以 COM 为基础，但在应用方式上则更多地继承了 MTS（Microsoft Transaction Server）的处理机制，包括 MTS 的对象环境、安全模型、配置管理等。COM + 把 COM、DCOM 和 MTS 三者有机地统一起来，同时也新增了一些服务，如负载平衡、内存数据库、事件模型、队列服务等，形成一个概念新、功能强的构件体系结构，使得 COM + 形成真正适合于企业应用的构件技术。四者之间的结构关系如图 4-22 所示。

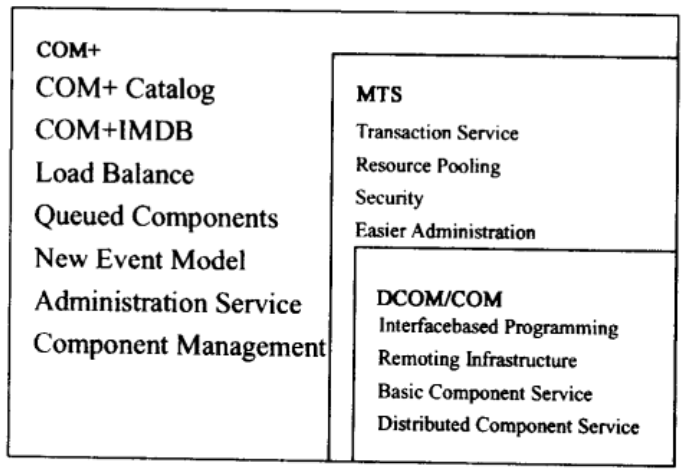


图 4-22　COM/DCOM/MTS/COM + 之间的关系

微软在 MTS 的基础上提出了多层软件结构的概念，如图 4-23(a)所示。

为实现多层结构的企业应用，需使用各种分离技术，开发人员费时费力。图

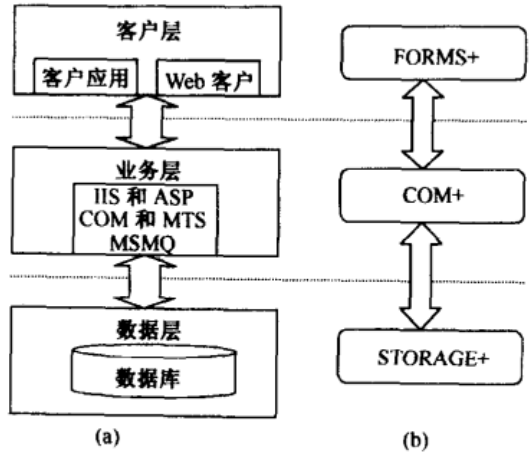


图 4-23 多层软件结构

4-23(b)则是微软在 Windows 2000 实现的 DNA 结构的 3 层结构模型。客户层 FORMS+ 还只是一个技术框架，它把 Win32GUI 和 WebAPI 结合起来，并朝着 DHTML 的方向发展。数据层 STORAGE+ 还只是一种提法，不过微软已经把数据库接口从 ODBC 转移到 ADO 和 OLDB 上，这将最终促进数据层接口技术的统一。中间业务层 COM+ 以系统服务的形式把原先散落的一些技术综合起来，并提供简单的编程模型，以直接应用层的编程接口为应用程序提供服务。COM+ 是 DNA 结构的核心，它将成为企业应用或者分布式应用的基本工具。COM+ 构件建立在 COM + 系统服务基础上，可避免底层繁琐的细节处理，既保证应用程序的可靠性，又使其更趋于标准化。COM+ 构件提供可管理、可配置的特性，在创建 COM+ 对象时通过截取（intercept）技术为其分配一个环境对象（context），利用环境对象的 IObjectContextInfo 接口可以访问到环境的属性信息。下面对截取概念的步骤进行说明：

①构件对象通过说明性属性指定一些基本要求。

②客户端调用 CoCreateInstance 函数时，COM+ 系统检查客户代码是否运行在与对象类兼容的对象环境中。

③如果客户代码运行环境与对象类所要求的兼容，那么就不使用截取技术，直接创建对象并返回对象的接口引用。否则 CoCreateInstance 函数切换到一个与对象类兼容的环境中，然后创建对象并返回一个代理对象。

④在以后的接口方法调用中，代理对象在调用前后作一些处理，以便方法的运行环境能满足要求。

COM+ 的对象引用即对象接口指针与环境相关，不能简单地把对象引用从一个环境传递到另一个环境。当客户从一个环境调用到另一个环境中的对象时，

中间必须经过代理对象和存根代码，由代理对象截取调用，负责进行环境切换，保证客户代码和对象分别在自己的环境中执行。类似于 COM 的跨进程列集（marshalling）和散集（unmarshalling）处理，即调用 COMarshalInterface 和 CoUnmarshalInterface 函数，对于支持事务特性、安全特性及其他特殊要求的应用较为重要。跨环境调用过程如图 4-24 所示。

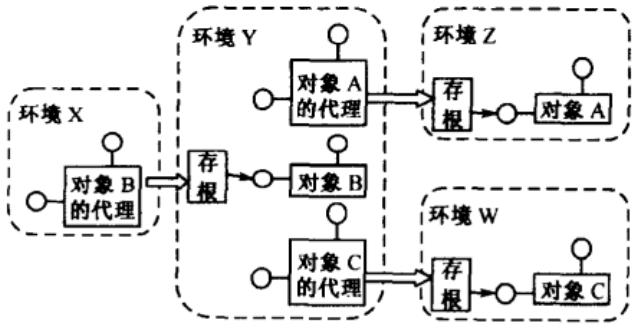


图 4-24　跨环境调用过程

COM + 最具特色的系统服务有的从 MTS 继承过来，如事务、对象池、安全模型以及管理特性，有的是新增加的，如队列构件、负载平衡、内存数据库和事件服务。COM + 以系统服务的形式提供应用有多方面的好处。其一是客户或者构件程序直接利用系统服务，避免底层细节处理，减少开发成本，降低编码量；其二是有些系统服务涉及到较复杂的逻辑，如需进行底层系统资源的访问，应用层较难实现；其三是使用系统服务可增强可靠性。

2．COM + 提供的各种服务

（1）COM + 目录服务

COM 和 MTS 构件把它们的配置信息放在 Windows 的注册表中，而 COM + 则把大多数构件的信息放在一个新的数据库里，这个数据库就叫做 COM + 目录（COM + catalog）。COM + 目录统一了 COM 和 MTS 注册表，并提供了一个构件管理环境。开发人员通过 COM + Explorer 或一系列新的 COM 接口来访问 COM + 目录。

COM + 的一个新特性是它支持声明编程（declarative programming）。意思是说，开发人员按通常的一个方式开发一个构件，在分发的时候构件的细节却各不相同。例如，开发人员开发一个可以在有负载平衡环境下工作的构件，但是否使用负载平衡特性却各不相同。有些应用程序需要使用负载平衡特性，而另一些不使用。通过 COM + Explorer 在管理层上的设置来决定是否使用负载平衡。

（2）COM + 的负载平衡

现在的 COM 和 MTS 的一个缺点是它们不支持动态负载平衡。为了创建远程构件的一个实例，客户程序必须显式地指出构件服务器的名称。这样就导致应用程序的扩展性不好。COM + 通过对客户端应用程序提供透明的负载平衡服务解决了这个问题。如图 4-25 所示。

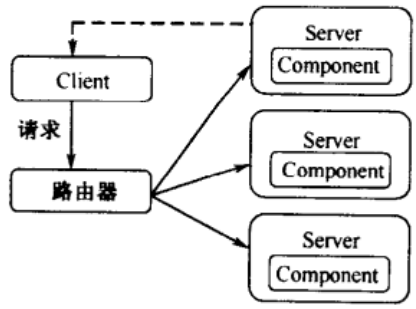


图 4-25 COM + 及其动态负载均衡

为使用负载平衡特性，首先定义一个由多台安装了构件的服务器组成的应用服务群，然后配置一台负载平衡路由器，由它接收创建对象请求，并将请求送到应用服务器群的某台服务器上。COM + 负载平衡作为一个 Windows NT 服务运行在路由节点上。当收到一个创建对象的请求后，它和本地 SCM（Service Control Manager）一起将请求送到应用服务器群中负载最轻的服务器上。

每个具有负载均衡能力的构件都与一个负载平衡引擎相关联，引擎和路由节点通信来决定请求送到哪一台服务器上。缺省的 COM + 负载平衡引擎使用应答时间算法，应答时间通过测量每台构件服务器上各个接口的每个实例的所有方法调用的时间计算出来。这个算法不一定对每个应用都最适应，但在大多数环境下工作的很好。程序员还可以开发自己的负载平衡引擎，引擎构件本身也是一个 COM + 构件。

路由节点在收到创建对象的请求时，首先在 COM + 目录检查该构件是否支持动态负载平衡，如果是，则将请求送到合适的服务器。请求被送到服务器后一旦对象被创建好，则将对象的引用直接传送到客户，在客户和服务器间直接建立联系。

虽然 COM + 负载平衡对客户端透明，但为了提高效率，在编程时仍要小心。首先，创建远程对象时使用路由节点的名称而不使用远程服务器的名称。由于 COM + 的动态负载平衡只在构件激活时起作用，所以客户应只在需要的时候创建远程对象，使用完毕后尽快的释放（这和 MTS 客户编程有点不一样。MTS 客户应尽早得到对象引用，然后就一直抓着不放，以免频繁的创建和释放对象）。

虽然 COM + 动态负载平衡仍然存在一些问题，比如说在路由节点上存在单点失效问题，但它毕竟在对分布式应用的扩展性方面迈出了一大步。

（3）COM + 内存数据库

COM + 的内存数据库结构如图 4-26 所示。

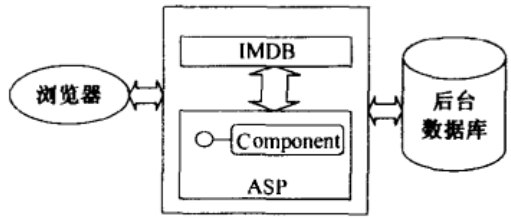


图 4-26　COM + 内存数据库

我们知道，对于数据访问频繁的应用程序，提高其性能的最好方法就是把尽可能多的数据放到物理内存。COM + 的 IMDB（In-Memory Database）就是提供了这方面的服务。IMDB 是只在物理内存上操作的临时性事务数据库系统。虽然 IMDB 主要是用于 Web 环境以解决大量用户对数据库的访问问题，但它依然适应于任何需要快速大量数据访问的应用。

IMDB 实现了一个面向数据库，优化查询的缓冲系统。它能从后台数据库装入物理数据或者用来存放临时数据。对一个热门 Web 站点，每分钟要对成千上万的用户提供数据查询服务，这个代价将是相当昂贵的。通过使用 IMDB，把那些访问频繁的表格装入 Web 服务器的物理内存，就能简单地通过增加物理内存来支持更多的用户，而内存的价格越来越便宜，这样也就大大降低了成本，同时也减少了 Web 服务器和数据库服务器之间的网络流量。

IMDB 的一个重要的不同是它不通过 SQL 语句来访问数据。IMDB 的主要目标是尽可能快地访问数据，为了达到这个目标，数据访问是通过标准的 ISAM 技术而不是 SQL 解释器来实现的，这就是说，必须对查询或过滤的字段建索引。

IMDB 作为快速数据访问的缓冲相当有效，它同时也能为应用程序管理临时数据。现在的 MTS 构件通过 SPM（Shared Property Manager）来共享临时数据。IMDB 现在支持事务操作，并且将来也会支持分布操作，它最终将会成为共享临时状态信息的解决方案。

（4）COM + 对象缓冲

对象缓冲（object pooling）是把多个构件实例装入内存的过程。对象缓冲后，当有客户请求到来时就能立即给客户应答，大大提高了客户响应速度。对象缓冲是大型可扩展应用程序的另一个重要特征。在 MTS 环境下开发构件时，构件实现 IObjectControl 接口提供激活/失效通知。当使实例无效时，MTS 调用 CanBePooled()来看实例是否能放入缓冲。现在的 MTS 不支持对象缓冲，所以 CanBePooled()永远不会被调用。COM + 系统则支持对象缓冲。

其实大多数情况下不需要开发对象缓冲的构件。那么，什么时候需要呢？一是对象的创建时间大于对象的实际使用时间；二是构件访问有限资源如数据库、管道连接等。由于对象缓冲的大小可以限制，所以对第二种情况特别有效。

COM + 系统已经提供了许多系统级的支持对象缓冲的构件（如 ODBC 数据源对象等），所以在开发大型的可扩展应用程序时开发人员可以集中精力提供应用级构件。

（5）COM + 排队构件

现在的 COM 开发模式是基于过程调用。客户与构件建立 RPC 连接，查询某个接口，然后用返回的接口同步调用某个方法，通过［out］参数取得返回信息。客户的生存期和构件实例紧密联系在一起。

COM 提供的基于 RPC 的服务对构件分布式应用来说是必需的，然而，对某些应用来说，使用消息传递技术可能更合适。新的 COM + 排队构件（queued component）服务为这一类应用提供了解决方案（如图 4-27 所示）。

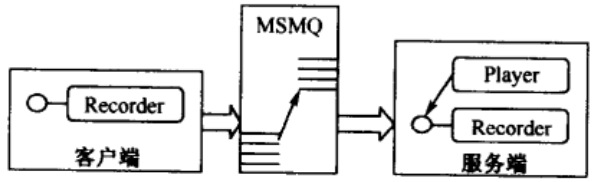


图 4-27 COM 排队构件

基于消息传递的应用在底层使用消息传递系统而不是 RPC 连接，这样，将客户和构件的生存期分离开来。COM + 排队构件使用的底层系统是 MSMQ（Microsoft Message Queue Server），客户和构件则通过排队机制分离开来。

这样就使应用程序具有更加灵活的可扩展性和可用性。如果构件暂时不可用，则客户程序仍然可以执行。如果客户很多而服务器很少，则可以将客户的请求予以安排，从而均衡服务器的负载。对像 Web 站点这样负载不稳定、起伏大的应用来说，采用消息传递方法将带来极大的好处。

当然，基于 RPC 连接和消息传递的服务对分布式应用来说都需要，开发人员可以根据最终用户需求来选择适当的技术。

像大多数 COM + 服务一样，COM + 排队构件尽可能多地将底层细节对开发人员隐藏起来。实际的排队机制对开发人员透明，开发人员可以像开发其他构件一样开发 COM + 排队构件，但有两点例外：一是构件接口只能有［in］参数，这些参数将通过 MSMQ 消息按值而不是按地址传递；二是构件接口方法不能返回值。

当客户实例化一个构件时，COM + 运行环境实际上在本地创建一个称为

“录音机”的代理对象。当客户方调用时，录音机把调用按顺序“记录”到MSMQ 消息里。客户释放对象的所有引用后，这些消息通过 MSMQ 被送到服务器。在服务器上，一个特别服务将这些消息取出来，然后用另一个称为“收音机”的代理生成构件实例。放音机“播放”那些方法调用与构件实例交互，就像构件同客户交互一样，如果出错，则将消息放回队列里面去。

COM + 提供的分离、异步处理使开发人员在开发分布式的企业应用程序时具有更大的灵活性。

(6) COM + 事件

COM 提供了两种技术来处理构件和客户之间的事件。第一种技术是使用接口回调机制，由客户实现服务构件描述的接口，构件服务器通过调用这些客户接口来实现事件触发。第二种技术叫做连接对象（connectable object），使用标准的 COM 接口 IConnectionPoint。第二种方法和第一种方法相似，但它提供了一种更通用的方法来连接客户和构件。

谈到事件的时候，客户和构件概念就模糊起来。事实上，构件就变成了客户，客户变成了构件。它们实际上是两个共享信息的合作软件实体。COM + 称发布信息的实体为“发行者”（publisher），接收信息的实体为“订阅者”（subscriber）。

COM 构件模型有以下缺点：

① 发行者和订阅者紧密联合在一起，在编译时互相要知道对方的接口定义。

② 支持多路广播要编写许多额外的代码。

③ 这种模式只定义了一些接口，开发人员仍然需要编写代码来实现这些接口。

④ IConnettionPoint 在分布环境下效率特别差。

⑤ 不支持持久连接，当然这也是由于 COM 的实现是基于 RPC 的。

COM + 事件模型（如图 4-28 所示）引入了一个称为事件类（event class）的中间对象。事件类是由 COM + 运行环境实现的构件，位于发行者和订阅者之间。因为事件类实现了事件接口，对发行者而言，它是订阅者。当发行者要触发事件时，它创建一个事件类实例，调用某个事件类接口方法，然后释放接口。运行环境决定何时以及怎样通知订阅者。像排队构件一样，发行者和订阅者的生存期是分开的。也是由于同样的原因，事件接口只能有 [in] 参数。

接收事件比较简单。订阅者通过运行环境的帮助创建一个订阅对象，并将它加到 COM + 事件库（COM + Event Store），这样当发行者触发事件时，事件类就能确定谁订阅了事件，并通知订阅者，甚至在必要的时候激活订阅者。

总而言之，新的 COM + 事件系统：

① 订阅者和发行者之间的生存期分离。

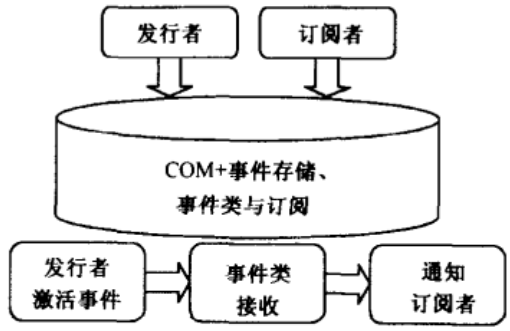


图 4-28 COM + 事件模型

② 在事件系统增加了一个激活模式。当事件发生时若订阅者没有激活，运行环境可以激活订阅者并将事件信息传给它。

③ 提供一个第三方的发行者 - 订阅者环境。一旦事件类创建，双方都可以是事件的发行者和订阅者。

④ 支持丰富的过滤机制。通过编写过滤对象，可以在订阅者级和发行者级两端过滤事件，后者的效率更高。

COM + 事件系统不但是为应用程序开发，它还在 COM + 内部被用来处理调试和跟踪信息，甚至用来实现操作系统本身。

4.3.6 CORBA 与 DCOM 的主要异同

(1) CORBA 和 DCOM 的相同点

① 都完全支持面向对象的范型，系统的软件元素以对象的形式出现，相互间的交互采用对象中的发消息形式。

② 两者提供的某些服务是相同的，如命名服务、对象的永久性存储服务、版本服务等。

③ 都支持多平台。

④ 都支持服务提供对象的自动激活，即当一客户对象请求某一服务时，若该服务未启动则自动将其激活。

(2) CORBA 和 DCOM 的不同点

① CORBA 规范不是针对特定厂商的，因此 CORBA 应用能运行于不同的硬件平台上。DCOM 则是由微软制定、拥有的体系结构，并且只能运行于微软操作系统支持的硬件平台上。虽然随后由 Software AG 将 DCOM 扩展到了其他的平台，但一些测试也证明了非 Windows 平台的 DCOM 在可靠性和性能方面不尽人意。

② CORBA 体系结构是基于对象请求代理的；DCOM 则以 COM 作为它的基

础，事务处理则依赖于 MTS 或 MSMQ。

③ CORBA 与其他协议的互操作性（连接性）好，如 GIOP（General Inter ORB Protocol）提供了不同 ORB 之间的连接，IIOP（Internet Inter－ORB Protocol）则是 GIOP 在 TCP/IP 网络上的实现，ESIOP（Environment Specific Inter－ORB Protocol）提供了 ORB 和 DCE 之间的连接。

④ CORBA 的大多数服务均由 OMG 自行定义，而 DCOM 的传输和安全服务则基于 DCE 的标准。

⑤ CORBA 中对象之间的通信机制为基于 ORB 的消息传递，依赖于 IIOP 进行远程对象通信；而 DCOM 则使用 ORPC（Object Remote Procedure Call），它是对 DCE RPC 的扩展，其中增加了一种新的数据类型，即对象应用类型。

⑥ 尽管 DCOM 的安全机制是平台相关的，但它非常灵活，基于 Win32 的安全管理器 API 已经可用，通过它可以调用 Windows Domain 授权机制，以及 Novell 和 DCE Kerberos 审计程序。与此相比，CORBA 的安全规格定义出台不久，各个 ORB 厂商所开发的安全性子系统各不兼容。

⑦ CORBA 支持多继承，DCOM 仅支持单继承。DCOM 的 IDL 中所有界面都从一个共同的类 IUnknown 继承而来，而 CORBA 无此共同的超类。

⑧ DCOM 每两分钟使用 ping 的方法检查客户是否依旧处于激活状态，如果客户超过 6 分钟时依旧未作响应，DCOM 则会取消该客户请求。相反地，CORBA 不强迫客户保持连接状态，并且不使用保持激活状态的通信方法。由于 DCOM 使用保持激活状态的通信方式，因此能够决定何时取消请求，从而使用内建的垃圾收集功能；CORBA 则不提供内建的垃圾收集方法。

总之，对于应用者来说，COM/DCOM 是微软拥有的体系结构，仅被 Windows家族的操作系统支持。不管怎样，有第三方厂商提供了在 Unix 系统上的 DCOM 支持。DCOM 基于自然的二进制格式，因此执行较快，但是不适用于其他平台。COM/DCOM 组件能够访问 Windows API，因此能潜在地损坏或危及用户的计算环境。DCOM 为分布式对象提供基本的支持，但不支持实时处理，也不适于在需高可靠性的情形下使用。虽然 COM 已经出现了很长时间，但是，对于它的扩展 DCOM，在大量的分布式应用中的前景，还不是很明朗的。而 CORBA 仅仅是一个规范，而不是一个实现，因此，用户很难确定所购买的产品是否完全兼容 CORBA。并且没有已定义的测试套件，来测试是否兼容 CORBA。对于客户来说，需要有行家对厂家的产品进行评价。CORBA 是一个复杂的规范，需要有相当多的专家来开发分布式对象和应用。从另一方面来讲，使用 CORBA，能使开发分布式应用变得容易。当然，需要很多对分布式系统设计，分布式、多线程程序设计和调试，内联网和面向对象分析、设计精通的专家。

对于分布式计算，COM/DCOM 和 CORBA 都具有可扩展性和健壮性的结

构，并且具有各自不同的优势。不管怎样，鉴于它们内在的区别，它们分别适用于具有不同规模和类型的应用中。如果系统主要运行微软操作系统，并且其地域分布上不是很广的话，那么，COM/DCOM 或许是比较合适的。CORBA 则适用于异构的、大规模的分布式系统。两种技术体系结构有其相同点和不同点，因此，在挑选产品时，必须作一番考虑。当然，许多开发商为 CORBA 应用和 COM 交互提供了许多解决方案。看上去，COM/DCOM 和 CORBA 将会继续强有力地竞争下去，并且会长久地并存。

参考文献

车文富.2001.CORBA 与 OLE/COM 互操作实现技术.计算机工程与应用,13(7):104～106

陈坚,林亚平.2000.CORBA 对象调用存储过程的应用技术.计算机工程与设计,21(2):30～34

黄庆荣,傅清祥.2001. 基于 CORBA/COM 互操作的 GIS. 福州大学学报 (自然科学版),29(4):53～56

来欣等.1999.CORBA 与 OLE/COM 的应用集成研究.计算机工程与应用,2(2):40～43

李天宁,魏明亮等.2001.CORBA3.0 新特性的分析及评述.计算机工程与应用,3(3):38～41

楼伟进,应飚.2000.COM/DCOM/COM+构件技术.计算机应用研究,20(4):31～33

吕钊,顾君忠. 2001. 关于 CORBA 消息机制的研究.计算机应用研究,8(8):64～67

潘登,侯文永.2000.COM+及其基于属性编程.计算机应用研究,5(5):54～57

潘爱民.1999.COM 原理与应用.北京:清华大学出版社

王鹏,尤晋元.1998.CORBA 与 DCOM 的比较.计算机工程,24(9):11～13

王振宇.1999.CORBA:全面的分布式对象计算(上).计算机工程与应用,20(10):27～29

王振宇.1999.CORBA:全面的分布式对象计算(下).计算机工程与应用,20(11):28～30

伍光胜,郑明辉,黄远铮.2001.COM/DCOM 技术的分析及应用.计算机应用研究,9(9):64～67

项君,高洪奎,吴泉源等.2001.CORBA 分布式计算环境中对象适配机制的优化及构件化.计算机工程与科学,22(5):28～31

张岩,周可记.2001. 中间件技术与应用研究.计算机与通信,1(1):31～38

张久文,李治柱,赵春云.1997. 基于对象请求中间件(ORB)的 Client/Server 接口.上海交通大学学报,31(3):83～88

周密,贾惠波.2001.CORBA 与 Enterprise JavaBeans 相结合构建分布式对象系统.计算机应用,21(8):27～29

Dirk Slama, Jason Garbis, Perry Russell. 1999. Enterprise CORBA. NJ: Prentice Hall

ITU/ISO. 1996. Reference Model of Open Distributed Processing – Part 1: Overview. ISO/ICE, ITU TRec. X901

ITU/ISO. 1995. Reference Model of Open Distributed Processing – Part 2 : Foundation. ISO/ICE, ITU TRec. X901

ITU/ISO. 1995. Reference Model of Open Distributed Processing – Part 3 : Architecture. ISO/ICE, ITU TRec. X901

ITU/ISO. 1995. Reference Model of Open Distributed Processing - Part 4: Sematics. ISO/ICE, ITU TRec. X901

ITU/ISO. 1997. ODP Trading Function in Part 1: Specification. ISO/IECIS, ITU/TdraftRec

OMG. 1995. OMG Common Object Service Specification Revision 2.0

OMG. 1998. OMG Common Object Request Broker Architecture and Specification Revision 2.2

OMG. 1998. OMG CORBA/IIOP 2.2 Specification. http://www.omg.org

Rogar Sessions. 1998. COM and DCOM: Microsoft's vision for Distributed Object. British: John Wiley & Sons, Inc.

Sanjeev Krishnan. 1999. Enterprise Java Beans to Corba Mapping, v1.1. SUN Microsystem

Schmidt. 1998. Using the Portable Object Adapter for Transient and Persistent. CORBA Objects, 12 (4)

Thomas Mowbray, Raphael. C. Malveau. 1997. CORBA Design Patterns. British: John Wiley & Sons, Inc.

Tom Armstrong. 1999. COM + MTS = COM +, Next Step in the Windows Component Stragety. Visual C++ Developers Jurnal, (2-3)

第5章　软件MAS技术

Agent一词最早可见于M.Minsky于1986年出版的Society of Mind一书。M.Minsky引入了Society和Social Behavor的概念。他指出，个体存在于社会之中，社会中的个体在有矛盾的前提下通过协商或者竞争的方法得到对问题的求解。这些个体被称为Agent。1994年，M.Minsky在CACM上对Agent的概念作了进一步的说明。他认为：Agent是一些具有特别技能的个体。对于计算机而言，Agent是指“当你试图说明完成一些任务的机器而无需了解它是如何工作时，即将其处理为黑箱时，就称其为Agent”。

对软件Agent的研究主要分为两条主线，一条主线是自20世纪70年代起围绕AI展开的研究，另一条主线是从20世纪90年代左右开始的以应用研究为主的研究。前者主要研究Agent的拟人行为和多Agent协作模型等，其内容可分为Agent理论、Agent体系结构、Agent程序设计语言、多Agent模型等。后者的研究结合了智能计算、人机界面和软件工程学的最新研究成果，发展成为软件代理技术。

软件Agent的研究已经在计算机科学的各个领域引起了极大的兴趣，并且已有的一些成果被应用到了各个研究领域。软件Agent是完成一些特定的任务，并具有自主性、协作性的计算机程序。对软件Agent的研究，分布式人工智能中采用了拟人的描述，即软件Agent是组成Agent社会的成员，Agent是包含了信念、承诺、义务、意图等精神状态的实体。而软件工程则从模型角度来考察了Agent，认为面向Agent的软件开发方法是为了更确切的描述复杂的开发系统的行为而采用的一种抽象描述形式，它与面向对象方法一样，是观察客观世界及解决问题的一种方法。可见，构造具有自主性、协作性及智能的软件Agent是研究Agent的最终目标。Agent已逐渐成为人工智能研究的中心议题，在计算机领域中已上升到重要的地位。软件Agent技术被人们称为是软件领域的一次重大革命。

软件Agent是一个飞速发展的领域。目前在国内，关于Agent一词有许多译法，例如：“代理”、“智能体”等，但是这些名词并不能全面反映一个Agent所具有的本质属性，所以本书中仍然采用Agent这种自然形式。Agent是一个交叉性研究领域，是在基于计算机科学、人工智能、并行计算、分布式系统、知识工程、专家系统等多个研究方向和领域之上的交叉性研究领域。

人们基于Agent对软件的研究主要可分为两类：关于单一Agent软件和多

Agent 系统。在前一个领域中，对单个 Agent 的设计研究可从 3 个层次进行：Agent 理论、Agent 结构和 Agent 程序语言。Agent 理论讨论什么是 Agent 以及如何用形式化的方式表示 Agent 特性并进行推理等问题；Agent 结构则关注如何设计同 Agent 理论特性相一致的软件系统等问题；Agent 程序语言则主要涉及适合 Agent 开发编程的有关语言的设计等问题。而关于多 Agent 系统的研究也可分为 3 个方面：多 Agent 的组织、多 Agent 系统的动态性和多 Agent 系统中 Agent 的社会行为。多 Agent 的组织方面主要包括 Agent 之间的合作关系和通信的研究；多 Agent 系统动态性的研究是有关 Agent 行为的一致性、Agent 之间的协调和谈判等方面；Agent 的社会行为则主要讨论 Agent 之间的推理、所处的软件环境评估等问题。目前对 Agent 技术的研究主要集中在以下几个方面：

① 面向 Agent 的程序设计技术 AOP（Agent-Oriented Programming），最早是由Y.Shoham 在 1993 年提出的，他认为 AOP 是一种以计算的社会观为基础的新型程序设计规范，并将 AOP 看作是 OOP 的特例。两者都是面向对象的程序设计方法（Agent 是一种特殊的对象），都支持消息机制。但两者的不同在于：Agent 作为对象本身具有心智状态、信念和能力等；另外 AOP 的消息是独立于 Agent 的，对于 MAS 中的每一个 Agent 而言具有相同的语义，而 OOP 中的消息是针对每一个对象而言的，同一个消息对于不同的对象而言可能具有不同的语义。

② Agent 特性的形式化研究。R.Goodwin 提出了一个 Agent 系统的抽象模型，并用形式化对该模型中包含的 Agent、任务和环境 3 个要素进行了定义，这对于开发实际应用系统具有指导意义。

③ 基于 Agent 的软件工程（Agent-based software engineering）。目的是为了解决同类或异类 Agent 之间的互操作问题。Gemesereth 等人提出了一种“联邦式”（federation）的 Agent 体系结构，并设计了一种用于 Agent 之间相互对话的通用 Agent 通信语言 ACL，通过 ACL 可以完成各种 Agent 之间的通信行为。

④ 有关 Agent 开发环境的研制是为了以工程化的方法提高 Agent 的开发效率，目前已有一些特定类型的 Agent 开发环境研制成功。如 M.H.Coen 开发的 SodaBot 是一个具有通用性的软件 Agent 用户环境和构造系统，D.C.Smith 等人开发的 KidSim 是一个工具包式的 Agent 软件开发环境等。

⑤ 关于 Agent 协商模型的研究是为了寻找多 Agent 系统环境中 Agent 之间避免冲突的有效途径。对它的研究侧重于这些 Agent 为了采取联合行动或解决各自问题，如何协调各自的知识、目标、策略和规划等内容。此外，针对特定领域的 Agent 协商模型的研究也正在进行之中。

在 Agent 领域的研究中，人们使用了许多相关的学科（如社会学、心理学、认知科学和协调理论等）的研究方法与结论。同样，Agent 技术的研究成果也广

泛应用于其他领域之中。

软件 Agent 技术为复杂的大规模软件构架设计提供了新的途径，许多复杂的软件系统无法通过分析设计完整的系统规范，然后使用传统的软件工程模式开发和实现。基于 Agent 系统的开发方法，要求构造一些复杂的能够自治的构件，并能够与其他一些独立开发的 Agent 灵活交互。Agent 之间的交互行为不再通过严格预定义的接口实现，而是通过 Agent 之间的协商来实现，这意味着系统的整体性质和行为不能预先在程序中固定，必须在运行时通过参与系统活动的各个 Agent的行为和交互作用来动态体现。

5.1 软件 Agent 的概念和 Agent 联邦

由于软件 Agent 的研究受到智能计算、人机界面和软件工程等领域相关技术的影响，使得软件 Agent 的含义也具有多重性。对于一个完整的 Agent 究竟应该有哪些特性，学术界和产业界迄今为止并没有一个一致认可的定义。虽然对 Agent的理解各人有不同的观点，但有一点是明确的，即软件 Agent 是具有自主性和协作性的计算机程序，它能够帮助用户完成一些特定的任务。目前，大多数软件研究人员认为 Y.Shoham 有关 Agent 的定义较为全面和准确，他从计算的社会性角度出发，提出用 Agent 社会来构造计算机系统以联合解决问题。

5.1.1 软件 Agent 的性质和定义

给软件 Agent 一个明确的定义形式是十分困难的。Carl Hewitt 在 DAI 研究报告中指出：什么是 Agent 的问题对于基于 Agent 的计算领域与提出什么是人工智能的问题是同样困难的。在此，首先讨论 Agent 的性质。

1. 软件 Agent 的性质

从本质上看，软件 Agent 应该具有下述 4 个本质的属性，即交互协作性、自主性、可控性和目标/任务驱动性。

(1) 交互协作性

Agent 能不断地监测周围的环境和其他 Agent 所发出的交互信息和服务请求。由 Agent 事件处理系统控制自身的行为，使其与其他 Agent 有效协同地工作，并能以类似人类的工作方式和人进行交互。软件 Agent 的交互协作性（interactive）包含两个方面的内容：Agent 与人的交互协作和 Agent 与 Agent 的交互协作。Agent 与人的交互协作常见于界面智能 Agent，这类软件 Agent 主要通过 Agent 与人的交互界面，对于用户提出的需求给以反馈，或帮助用户完成复杂、费时的任务，它们一般具有学习功能，即能够模仿用户的操作习惯或者满足用户

的兴趣爱好。另一种交互是 Agent 与 Agent 的交互协作，这类经常存在于多 Agent系统 MAS（Multi－Agent System）中，不同的 Agent 之间通过协商或者竞争方式达到对问题的求解。一般地，完成 Agent 与 Agent 之间的交互协作需要具备 3 个内容：Agent 定义语言、Agent 通信语言和 MAS 系统架构。目前已经有了多种 Agent 的定义语言和通信语言。在 MAS 体系结构方面，主要包含两种方式：直接通信方式和协调通信方式。在一个 MAS 中，一个 Agent 所能实现的目标往往是 MAS 目标集的一部分，因此它必须与其他 Agent 合作，通过信息共享、观点的互相激励协商、帮助群体聚焦于待求解问题最相关的信息等手段最终来共同实现目标。

（2）自主性

软件 Agent 的自主性（autonomous）是指软件 Agent 可以在不需要人为直接干预的条件下，在协同工作环境中独立自主的行为实体。Agent 能够根据自身内部的状态和外界环境中的各种事件来调节和控制自己的行为，使 Agent 能与周围环境更加和谐地工作，提高工作效率。Agent 能适应动态变化的信息世界，独立自主地完成某些任务，为用户提供一定的服务。这意味着软件 Agent 具有知识和能力。软件 Agent 所具有的知识一般以知识库的形式表示。知识库中的知识一般包括 Agent 自身状态、运行环境状态、推理知识和动作知识等。在这里，将软件 Agent 的推理机制分为两个部分：即知识和能力。推理机制中的知识包括软件 Agent 进行推理所需要的状态知识、规则知识、判断知识等，而能力则指的是软件 Agent 根据已有的知识和状态进行推理的能力。一般而言，Agent 大都具有学习能力，即知识库可以变动，能力可以增强或者减弱。

（3）可控性

软件 Agent 的可控性（controllable）是指状态的可控。由于软件 Agent 的学习能力，软件 Agent 是否可能变得不受控制呢？软件 Agent 由于其实现手段和运行硬件环境等因素，可能在某些方面变得聪明，如软件 Agent 可以解决复杂的规划问题等，但从软件 Agent 本质上来讲，它所具有的知识和能力均基于客观形式的内在状态和外界环境状态，显然这些状态是可控的。

（4）任务/目标驱动性质

任何一个软件 Agent 都是为了满足用户完成某一类有用的任务而设计和实现的。它的基本运行机制为任务/目标驱动（task/goal driven）。这与传统的程序设计思想有所不同。在传统的程序设计中，较为常见的是数据驱动或者消息驱动。数据驱动最常见于管理信息系统，系统所涉及的一切操作或者动作均是由于数据而引发，系统本身也可以表示为一个或者多个数据流网络。消息驱动则最常见于 Windows 操作系统及其应用，系统中的功能或者动作均是基于对消息的监控和激发。而在软件 Agent 的设计和实现中，则是以任务/目标为基础，软件 Agent 的

动作始于用户的需求。关于目标/任务驱动与消息驱动、数据驱动的关系见图 5-1。

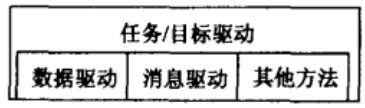


图 5-1 目标/任务驱动与数据、消息驱动的关系

在图 5-1 中，软件 Agent 所采用的任务/目标驱动方式是基于数据驱动、消息驱动和其他运作方式之上的，它可能采取数据驱动和消息驱动或者其他运作方式来实现。但是软件 Agent 对于系统的目标是作为整体来考虑的，即软件 Agent 利用知识和用户干预对目标或者任务进行分析，将之分解为子目标或者子任务，通过对子目标/子任务的求解进而完成整个目标/任务的求解。在分解目标/任务时，软件 Agent 才关心其内部是用数据驱动还是消息驱动方式来实现的。

(5) 软件 Agent 的其他性质

除上述之外，软件 Agent 还有其他许多性质，这与软件 Agent 的分类有关。

① 主动性。Agent 能自觉承担某些事件，并能遵循承诺，利用内部知识和能力持续主动地产生面向静态或动态目标的行为，直到完成整个任务。

② 独立性。Agent 可以看成是一个“逻辑单位”的行为实体，如同面向对象思想中的类概念，对其内部状态的信念、事实、过程及通信信息进行封装，使 Agent 成为协同系统中界限明确、能够被独立调用的计算实体。

③ 代理性。若当前内部状态和周围事件适合某种条件，Agent 就能代表用户有效地执行相应的任务，Agent 还能对一些使用频率较高的资源进行“封装”，引导用户对这些资源进行访问，成为用户通向这些资源的“中介”。例如，Agent 能根据用户自身的特点、爱好、习惯等自动代替网络用户寻找、收集，存储一些对用户有用的信息资源，加强了用户信息的完整性、全面性和实时性。此时 Agent就充当了人类助手的角色。

④ 反应性。指 Agent 利用其事件感知器感知周围的物理环境、信息资源、各种事件的发生和变化，并能够调整自身的内部状态，作出最优的适当的反应，使整个系统协调地工作。Agent 也能对突发事件作出相应的反应。

⑤ 智能性。Agent 根据内部状态，针对外部环境，通过感知器和执行器执行感知—推理—动作循环，这可通过人工智能程序设计或机器学习两种方式获得。

⑥ 继承性。沿用了面向对象中的概念，对 Agent 进行分类，子 Agent 可以继承其父 Agent 的信念、事实和属性等，如 Agent 牛可以继承 Agent 动物的所有属性。

2．技术特点

软件 Agent 技术可以看作软件开发的又一重大突破，这是因为它能够以一种更加自然、高效的方式解决传统软件业中存在的问题，而且为解决目前软件发展中其他技术难于解决的问题提供了一个理想的途径。与其他软件技术相比较，软件 Agent 在解决传统软件业中存在的问题时具有如下优点。

(1) 系统的数据、控制、专家知识和资源可以分布存储和处理

采用 Agent 技术使数据资源处于分布控制状态，数据的处理通常在本地进行，只需交换少量高层信息，因而提高了系统效率。

(2) Agent 系统可以自然地描述

对 Agent 组成的系统可以进行自然的描述，有利于各有关人员对软件系统的理解。比如在虚拟现实系统中，人物或其他角色可以自然地表示为自治的、具有个性化和社会性的 Agent。

(3) Agent 软件系统具有灵活性，有利于现有软件的集成

从长远的观点来看，维护旧系统，使它们能够继续发挥作用的最好途径是将它们包装成 Agent，使新的软件能够与其交互作用，利用它所提供的功能完成问题求解任务。

(4) 开放性和复杂性

Internet/Intranet 的发展对软件的开发提出了新的问题，目前软件生产所面临的难题是系统具有高度的开放性和复杂性，而软件 Agent 为解决这两个问题提供了独特的思路：

① 开放性。在以网络（Internet/Intranet）为中心的计算环境中，用户预先无法得知与其交互的各种资源情况，这就要求软件系统必须具有高度的灵活性和坚固性。因此，软件必须具有高超的社交能力，有助于达成一致的协商能力，控制协同工作的协调机制等。开放性还要求软件系统能够不断地监控它们所处的环境，并在适当时机产生新的目标。

② 复杂性。问题领域的广泛性、繁杂性和不可预测性，使得通用应用系统的开发是不可行的，惟一的解决途径是开发一些能够解决特定应用领域的专用模块化构件。软件技术的发展表明，一个大的求解问题可以分解成许多小的、简单的问题。采用分解策略的 Agent 软件使得系统的各个部分可以采用最适合的方法解决自己特定的问题，而不必采用统一的开发模式使整个系统折衷求全。面向 Agent 的系统分解较传统软件工程中的系统分解方式而言，能够通过其社交能力以灵活的、上下文有关的方式，而不是通过一些预先严格确定的接口函数，与外界进行交互作用。

3. 软件 Agent 的定义

由此可见，软件 Agent 可形式地定义为：软件 Agent 是运行于计算机上为用户完成有用任务的软件实体（对象），它必须具有以下本质属性：交互协作性，目标/任务驱动性，自主性和可控性。凡是满足上述条件的对象均可以称为软件 Agent。

5.1.2 软件 Agent 的联邦结构

Agent 作为一种能代表用户去执行计算和信息处理任务的智能化软件实体，一般都具有社交和领域知识，能依据心理状态（信念、期望和意向）自主工作，并具有语义互操作和合作的协调能力。

Agent 联邦（Agent federation）由管理 Agent（manager Agent）和其他成员 Agent（member Agent）组成的集合构成。其中，管理 Agent 对内负责本联邦内任务的调度、规划和分配，对外负责与其他 Agent 联邦（或单一 Agent）的通信，管理 Agent 是 Agent 联邦的核心。成员 Agent 在管理 Agent 中注册，接受管理 Agent 分配的任务并加以执行。

在软件 Agent 互操作系统中，通常由 Facilitator 和其他 Agent 构成联邦结构（federation architecture），如图 5-2 所示。

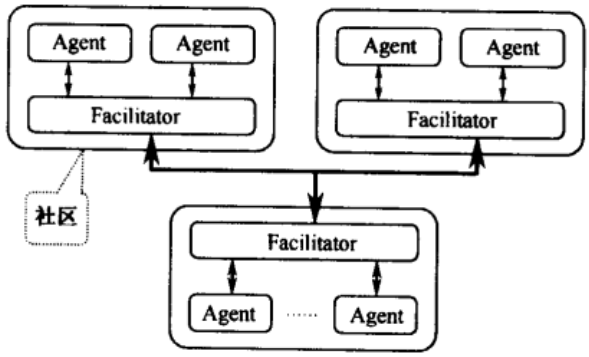


图 5-2 联邦结构

其中，整个系统被划分为一个个的社区（community），每个软件 Agent 分别属于一个社区，每个社区都有且仅有一个 Facilitator。在联邦结构中，只有 Facilitator可以和其他社区的 Facilitator 通信，其他软件 Agent 则只能同本社区的 Facilitator 通信，而 Facilitator 则完成将消息发往目的地的任务。事实上，Facilitator 所提供的服务远不止如此，Facilitator 存在的真正目的是隐蔽资源的物理分布，即每个 Agent 无需知道通信的对方是谁，只要其能提供对应的服务即可，这

为分布环境下的软件集成带来诸多便利，整个系统中多个 Facilitator 相互操作，向每个软件 Agent 提供一个内容丰富的服务网。由此可见，在 Agent 联邦中，Facilitator 有点像中间件，为 Agent 之间的合作和通信提供透明性。

通常情况下，Agent 之间的纵向合作发生在 Agent 联邦内部；而 Agent 之间的横向合作发生在 Agent 联邦之间，由各联邦的管理 Agent 相互协商确定是否参与合作。

5.2　软件 Agent 的分类

软件 Agent 一般认为是“具有 Agent 属性的程序”，就是说软件 Agent 是一种特殊的程序。软件 Agent 和常规程序的关系简单地说是动态与静态的关系、能力和知识的关系、整体和部分的关系。

目前，已经有许多软件 Agent，如何对其进行合理的分类是一个十分困难的问题。综合 Hyacinth S Nwana、Woodridge 和 Jennings 的观点，依照软件Agent的性质，将软件 Agent 划分为慎思 Agent（deliberative Agent）、反应 Agent（reactive Agent）、移动 Agent（mobile Agent）、学习 Agent（learning Agent）和混合 Agent（hybrid Agent）等。

1．慎思 Agent

慎思 Agent 最早出现于 Genesereth 所使用的 Deliberate Agent 一词，而后由 Mc Carthy 演绎使用为“A sentential processing automation or deliberative Agent”。慎思 Agent 一般包含符号世界模型，并通过模式匹配和符号逻辑推理来实现。慎思 Agent 主要面临以下问题：

①表示问题：如何使用符号准确合理地描述复杂世界；

②推理问题：如何利用符号逻辑推理进行决策；

③常识推理：事实上，一般情况下，规划是不可判断的，对于常识推理，要实现起来非常困难。符号演算的复杂性更使得较为简单的定理证明也要花费大量的时间。

规划 Agent 是较为重要的一类慎思 Agent。自从 20 世纪 70 年代以来，在 AI 领域，一般认为 AI 的规划系统是任何人工 Agent 的关键部件。规划本质上是自动程序，也就是一组动作的设计，使得执行时能够达到预期的结果。最早的规划器是 STRIPS。STRIPS 建立了世界的符号描述和目标的符号表示，以及动作的集合。每一个动作都对应着一系列条件。当执行时，STRIPS 则需要寻找一个动作的序列，通过双向匹配最终达到目标状态。在此之后，又出现了分层规划器和非线性规划器。

2．反应 Agent

反应 Agent 最早可见于 Brooks 和 A.Chapman 的研究。Maes 则提出了反应 Agent 应该具有的基本属性。反应 Agent 不包含任何符号世界模型，也不使用复杂符号推理。相反地，反应 Agent 直接以刺激/响应的方式进行运作和给以反馈。反应 Agent 结构设计来自于下面的假设：Agent 行为的复杂性是运作环境复杂性的反应，而不是其内部复杂设计的反应。反应 Agent 一般应用于游戏和系统模拟上。目前，反应 Agent 所面临的问题包括：反应 Agent 实现方法和语言，反应 Agent 适应能力和学习能力等。

3．移动 Agent

移动 Agent 是指可以在异构网络中自主的从一台主机漫游到另外一台主机，按照用户需求为用户完成有用的任务，并将结果以一定的格式返回给用户的一种软件 Agent。移动 Agent 的显著特点是从客户代理能够迁移到服务器代理上，与之进行本地高速通信，这种本地通信不占用网络资源。移动 Agent 一般用于建立网络索引、检索网络资源，或者帮助用户管理 E-mail、传真、电话等其他有用的任务。关于移动 Agent 的关键问题是它们的安全性、鉴证机制以及用户隐私等问题。

由于移动 Agent 在大规模软件构架中具有其突出的优越性，在降低网络瓶颈、增强系统的灵活性等方面独具特性，所以在“数字城市”的软件体系结构中具有重要的应用价值，有关细节内容参见以下章节。

4．学习 Agent

所谓学习 Agent 是指软件 Agent 能够通过学习的方法增长知识和能力，也就是说，软件 Agent 本身的知识库可以扩充，能力可以增强或者更加适应于用户习惯和兴趣等。学习方法一般包括经验学习、事例学习、概念学习、类比学习、神经网络学习等。学习 Agent 的典型应用是能够根据用户的兴趣自动提取网上新闻等资源，还能够满足用户检索领域相关资源等需求。E-mail Agent 是另一个学习 Agent 的例子。学习 Agent 的关键在于学习算法和评价等方面。

5．混合 Agent

许多研究认为：一个纯粹的 Agent 是不能满足实际需要的，即需要将几种 Agent 结合起来，互为补充，形成混合的 Agent 来完成特定的任务。混合 Agent 能够对一些复杂的问题进行规划和求解，也能对环境刺激以合乎常理的行为作出反应。目前，已经出现了多种较为著名的基于混合式 Agent 体系结构的Agent和

MAS，比如：Georgeff 和 Lansky 提出的 PRS（Procedural Reasoning System）；Ferguben 提出的 Touring Machine；Muller 提出的 InterRep 等。

此外，还有如下几个重要的 Agent。

6．用户接口 Agent

用户的智能接口，负责系统与用户之间的信息交换，帮助用户向系统提出要求并以用户喜欢的方式显示返回处理结果。用户只能通过用户接口 Agent（interface Agent）同系统进行交互。用户接口 Agent 确定一项任务后，即将任务送到由代理 Agent 推荐的最能满足用户需要的任务 Agent，由任务 Agent 完成相应的任务。

7．任务 Agent

任务 Agent（task Agent）从其他 Agent 那里得到要求解决的任务，确定所要达到的目标，并安排实现目标的计划，协调相关任务 Agent 或资源 Agent 共同完成任务。对于复杂的任务，它可进一步分解成子任务，交给其他 Agent 完成，最后集成各子任务返回的结果。当系统增加新的功能时，只需增加相应的任务 Agent。新增强的 Agent 向代理 Agent 宣传它的能力，其他 Agent 就可通过代理 Agent找到它。

8．资源 Agent

资源 Agent（resource Agent）对信息资源中的信息内容进行提取和更新。它实际上起到本地数据资源与系统中其他 Agent 之间的接口作用，作为其他 Agent 访问资源的中间媒介，它隐藏了本地数据的组织和表示。当有新的信息资源需要增加到系统中时，只要增加相应的资源 Agent，并使用 KQML 向代理 Agent 宣传它的服务内容。

9．代理 Agent

代理 Agent（broker Agent）为其他 Agent 提供语义比较服务，它比较哪些 Agent 提供的服务能够满足其他 Agent 请求的服务。

5.3 软件 Agent 的基本结构和工作机理

Agent 是一个具有信息处理能力的主动实体，具有与外界交互的感知器、通信机制，对信息进行存储加工的信息处理器、记忆库，同时还应该具有根据共同目标及自己的职责所产生的目标模块及反作用于外部环境的效应器，其基本结构

和工作机理如图 5-3 所示。

Agent 所采取的一切行为都是面向目标的。目标以任务表的形式来表示，可以动态地改变。任务表只是指明 Agent 必须做的事，并不是怎样做。任务的实现由效应器与其他 Agent 合作的形式来完成。

通信是交互的手段，由通信原语及通信内容两部分组成，它发送任务，表达各 Agent 对任务的态度及传递被处理的信息。

感知器表示 Agent 的感知能力。当触发事件被满足时，它被激活并接收外部信号/信息流。Agent 有两种行为：认知行为及效应行为。内部执行机制实现认知行为，它们改变 Agent 的内部状态及 Agent 的知识库，其他 Agent 感觉不到认知行为的执行。效应行为被效应器执行，效应行为的执行改变系统的状态，并且其他 Agent 能感觉到。

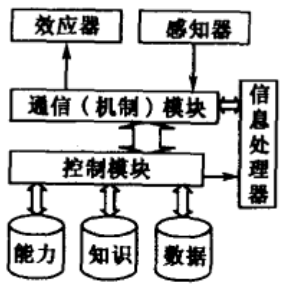


图 5-3 Agent 的基本结构模型

Agent 的信息处理器是体现 Agent 智能行为的最重要的部件。Agent 的信息处理器由信号/信息过滤器、控制器、符号推理机制、类比匹配机制、内部执行机制及知识库组成。其中，知识库中包含两类知识，一类是规则，另一类是知识块。信息处理器在接收到信号/信息后，先对其进行过滤、抽象、聚合，使其形成可以与客观世界的对象联系起来的有意义的符号，然后由类比匹配机制将这些符号及特征与知识库中的知识块进行模糊匹配。如果能查找到高度匹配的知识块，相应的知识块被用来处理信息并产生决策；如果只能部分匹配，则控制器驱动内部执行器将被匹配的部分知识作为符号，然后运用推理机制及知识库中的规则处理信息，并形成新的知识块；如果的确没有可用知识块，则知识库中的规则被用来处理信息。知识库中的规则及知识块随着问题的处理不断被添加和更新（见图 5-4）。

软件 Agent 的执行分为两个阶段：初始化阶段和消息循环阶段，典型的软件 Agent 的框架描述如下：

```
Agent () {
    Initialize ()              /* 完成初始化动作 */
```

```
While (! Done) {
    GetMessage (&msg);      /* 读一条消息 */
    ParseMessage (&msg)     /* 分析 */
    Do (&msg)
}
reture (Done);
}
```

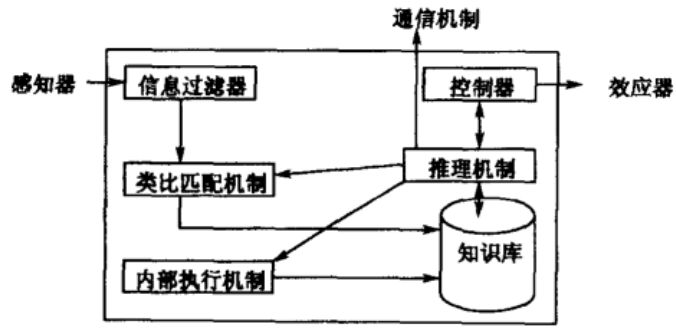


图 5-4　信息处理器结构

其中，Initialize()完成 Agent 在 ANS（Agent Name Service）上的注册，向本社区 Facilitator 发送兴趣文档等工作。对于不同的软件 Agent，由于其实现的功能与应用领域相关，因此只能仅给出一般的框架描述，具体的功能因具体领域而异。

5.4　移动 Agent

以上已经介绍过，移动 Agent 是一个能在网络上移动并完成相应任务的软件实体。由于它是一种非常重要的 Agent，在大规模软件构架中更有其独特的应用价值。

5.4.1　移动 Agent 的构成

移动 Agent 必须包含若干模型：代理模型、生命周期模型、计算模型、安全模型、通信模型和导航模型。下面从移动 Agent 的这些本质特点出发，对它加以精确的描述：

① 代理模型定义了 Agent 内部结构的一部分。如定义了自治性、学习性和主动性。

② 生命周期模型定义了移动 Agent 的各种执行状态和触发状态迁移的事件。

③ 计算模型定义了处于运行状态的移动 Agent 的执行过程。

④ 安全模型定义了移动 Agent 的安全机制。包括保护主机节点免受移动 Agent的侵害和保护移动 Agent 免受主机的侵害两个方面。

⑤ 通信模型负责 Agent 之间的合作，通常采用协议来保证通信模型的实现。

⑥ 导航模型涉及移动 Agent 的方方面面，主要包括从目标机的发现与决策到移动 Agent 的传送方式等。

移动迁移的内容包括代码和运行状态两部分。运行状态又可包括执行状态和数据状态。执行状态主要是指移动 Agent 当前运行的状态，如程序计数器、运行栈内容等。数据状态主要指与移动 Agent 有关的数据堆的内容。按所迁移的运行状态的内容，移动 Agent 的迁移可分为强迁移和弱迁移。强迁移同时迁移移动 Agent 的执行状态和数据状态，但这种迁移的实现比较复杂。弱迁移只迁移 Agent的数据状态，其速度较强迁移快，但不能保存移动 Agent 的完整运行状态。

移动 Agent 必须在特定的环境中迁移，这种环境称为移动代理环境。移动代理环境是分布于异构计算机网络上的软件系统，其主要任务是提供移动 Agent 的执行环境。该环境实现了移动代理定义中的大部分模型。除此之外，还提供了与自身的建造环境相关的支持服务、访问其他移动 Agent 的支持服务以及非 Agent 软件环境的开放性支持等。

5.4.2 移动 Agent 技术的优点

对于大部分采用移动 Agent 技术的系统，都可以运用现有技术构造出一个代替物。然而在某些特定的情况下，移动 Agent 在设计、实现以及执行等方面与传统的方法相比较有着不可替代的优点：

① 高效性。移动 Agent 把计算移至数据区，而不是把数据移至计算区，因而减少了对网络资源的消耗，提高了效益。

② 减轻网络负担。利用移动 Agent，用户可以把信息打包后发至目的地，在目的地进行本地交换，从而不再使用网络频繁地进行远程交换。

③ 异步式自主交换。系统把任务加载到移动 Agent 中发送出去，移动 Agent 独立于发送程序异步操作。

④ 实时性。对于实时性强的环境要作出及时的响应，如果能从中央系统发送移动 Agent 到本地来控制这些系统将大大减少响应时间。

⑤ 动态适应性。主要是指移动 Agent 对移动 Agent 环境变化的及时响应。

⑥ 处理大量数据的能力。要对大量的远端数据进行处理时，包含了处理功能模块的移动 Agent 可以移至数据存储地进行本地处理。

⑦ 透明性，即支持异构环境。由于移动 Agent 具有独立于计算机网络的特性，它支持透明操作。

⑧ 服务器行为个人化。在智能网络领域内，Agent 技术被认为是一种使网络

实体个人化的有效途径。

⑨ 方便的开发模式。移动 Agent 具有分布的本质，它的使用可以使大规模分布式系统的设计和构造变得相对容易。

5.4.3　移动 Agent 实现移动性的方式

Agent 的移动性通常有两种实现方式：远端执行和 Agent 迁移。

在远端执行方式下，Agent（程序代码 + 数据）被传送到远端系统，在远端系统中作为一个整体被激活和运行。任务完成后，Agent 可以自行销毁或存留在服务器上，转入休眠状态，以备以后某一时刻被激活。

Agent 迁移可看作是远端执行方式的一种扩展。在 Agent 迁移方式下，Agent不仅具有代码和数据，还有执行状态。它能在某一网络节点上暂时挂起自身的执行，迁移到另一个节点后，再从挂起前的状态继续执行。因而，在 Agent 移动方式下，一个任务可能经过多个网络节点后完成。

5.4.4　移动 Agent 系统实现的技术难点

① 克服计算环境的异构。移动 Agent 很可能在不同的环境中自主地执行，因此必须首先解决移动 Agent 的跨平台问题。

② 实现 Agent 的自主移动。Agent 的自主移动应解决以下 3 个问题：第一，Agent 的移动规程，包括 Agent 移动的触发、目的地指定、Agent 重新执行入口的指定等；第二，Agent 的通信模型；第三，Agent 的迁移方式。移动 Agent 在运行的过程中可能会因为本身的需要或意外事件而停止执行，需迁移到另外的站点上继续执行。

③ 保证移动 Agent 的安全性。它涉及 3 个方面：移动 Agent 自身的安全；移动 Agent 之间通信的安全和站点的安全。

④ 提供灵活方便的移动 Agent 环境。移动 Agent 系统用户的需求千变万化，对应的移动 Agent 也极其不同，因而用户必须根据自己的需求开发合适的移动 Agent，为用户提供一种方便的移动 Agent 编程语言及相应的开发环境等，都是移动 Agent 系统所要解决的问题。

5.5　软件 Agent 同专家系统和常规程序的比较

5.5.1　软件 Agent 与专家系统的比较

专家系统作为人工智能应用研究最活跃和最广泛的课题之一，现已在各个领域取得了很大的成功，其主要组成部分包括知识库、动态数据库、推理机、解释器和接口界面等。知识库存储关于某个领域的专门知识，推理机依据一定的策略

进行推理，动态数据库用于存放系统运行过程中所需要的和产生的各种信息，解释器负责解释用户需要了解的一些问题，接口界面则用于人机对话。

专家系统是目前人工智能发展中一个重要的研究领域，但是，传统的专家系统在实际应用过程中，有许多不完善的方面。

① 专家系统与用户的个性特征相分离。传统的专家系统不能适应用户的思维习惯、知识结构、学习和工作的方式、兴趣、爱好、专长等，即不能按照用户所熟悉的方式求解问题。

② 专家系统与应用场所的文化背景相分离。专家系统不能沟通应用场所的背景信息。专家系统在实际使用中，其所使用的语言、执行的动作都必须适合当时的应用场所。

③ 单个专家系统完成的任务是极其有限的。当需多个 Agent 协同工作才能完成一共同目标时，单个专家系统需要和其他 Agent 协调，任务的完全自动化可能不是最好的选择，因为求解非结构化的问题需要人类的智慧，因此更好的方法是自动化一部分，辅助一部分，人完成一部分。

④ 用户的地位。专家系统主要针对初学者，帮助缺少经验的用户解决一些问题，对于一些难于求解的问题最为有效。而对于有经验的专家，专家系统的作用就不很明显。Agent 系统中，将用户视为有相当能力的 Agent，由于有了人的参与，即使很难由计算机解决的问题，或者是不需要计算机做决策的问题，Agent系统也能解决一些非结构化的问题。

⑤ 专家系统是一个自治的问题求解器，用户是在完全被动的情况下，通过回答一些简单的问题就可以启动求解过程。专家系统的一个显著特点是求解的不精确性。专家系统适合于处理来自外部的信息是不完备的、模糊的，对所采纳的知识是经验的、不严格的场合。在 Agent 系统中，用户视 Agent 为合作求解问题的助手，机器通过 Agent 与用户一起讨论问题，交流信息，并完成对问题的求解。Agent 通过观察用户求解问题的过程和步骤，可以模仿进而代替用户完成一些重复的任务。

在专家系统的开发过程中，起决定作用的三方面因素是：领域专家知识的获取、表示与使用。传统软件侧重于数据处理，而专家系统着重于处理从领域专家那里获取的知识。知识表示的恰当与否，会影响知识获取能力与知识运用效率及推理机构的推理效率。

5.5.2 软件 Agent 与常规程序的比较

(1) 常规程序不具有智能性

人们习惯于把特殊求解的问题分解为若干个子问题，然后将分解到的子问题再分解，如此下去，直到子问题成为可以把握的对象为止，最后还原为待求解的

问题。这就是我们通常所说的结构化方法（或分析方法）。但智能问题是非结构化的，从这个意义上讲，用结构化方法编写的“智能”程序是让人怀疑的。因为这样的程序在同一参数下，无论你执行多少次，其结果是相同的。

在认识论上，分析法的思想主要表现为将新性质或新能力简单地归结为部分性质或部分能力，即还原论。不可否认，这种方法是一种重要的思维方法，对一个未知对象的研究、分析的重要性是有目共睹的。然而，它不是万能的，正是分析法的过度使用，使人们在研究一类非结构化的问题时，造成了认识上的混乱和困难，事实证明，分析法在 AI 研究中的使用是使 AI 陷入困境的根本原因。

(2) 软件 Agent 具有智能性

在软件 Agent 系统中使用了综合方法，从而使其具有智能性。Agent 的本质属性和 Lenat 对智能的描述是一致的，所以具有智能是不容质疑的。综合思想的智能性主要体现在以下几个方面：

① 多信道接收信息。人们的自身能力有限，却能自如地解决那些看起来难以处理的问题，并能作出合理的决策和反应，一个重要的原因是人们可以通过多种信道（如眼、耳、皮肤、舌等）获取各种信息。综合利用各种信息，采取逐步尝试的方法达到合理的目标，得到有意义的解。在计算机中的 NP 难题，在人类看来根本不是难题。给计算机装上录音机、摄像机、传感器等，计算机完全可以具有多信道接收信息的功能。

② 自动获取方法。人的知识有限，但人类对问题的求解方法却是多种多样的，在进行问题求解时，最主要的是将这些方法综合到一起，组成一个复杂的推理过程，虽然人的能力表现形式很多，如听力、视力、记忆力、想象力等，但这些能力不是孤立的，而是相互依存，相互制约。仅有知识的 AI 系统不是有能力的系统，重要的是有运用知识求解问题的方法。

③ 利用常识。我们知道，人类拥有大量的常识知识，人们会在求解问题时不知不觉地应用常识知识，这些广泛的常识知识的应用，往往能极大地约束人类的思考范围，使推理加速收敛，领域知识与常识知识的综合，在人的大脑中体现了“智能”的整体性。Lenat 的 CYC 工程是常识知识利用的典范，由于常识巨大，CYC 取得了一些惊人的成功，当然 CYC 还处在一个非常初级的阶段，而 Agent具有常识利用机制。

④ 系统集成。将已有的 AI 系统集成在一个系统中，使它们相互联系，相互制约。根据人行为的社会性，MAS（Multi-Agent System）的集成能力，使得软件 Agent 获得了整体综合效应。

⑤ 模式识别（非搜索推理）。对一个特定的字母（如“A”）无论是大是小、是正是斜、是倒是草、是标准还是不标准、有否背景噪音等，人们能轻而易举的识别它们，甚至损坏的“A”也能被辨识，这决不是搜索，而是整体的模式识

别，相当于人的形象思维，这是综合法较难使用的步骤。

⑥ 类比归纳。如果 A 和 B 看起来有些无法解释的相似之处，那么值得费时间去搜索其他的共性。这是人类具有创造能力的源泉，类比和归纳涉及当前情况与另一情况的部分匹配，这里有两个相互独立的方案：一是纵向变换，遇到复杂情况时，同一简单的情况做类比说明。二是横向变换，跨领域映射较少，但能带来好处。类比归纳渗透在人的交流中，似乎不合推理，但却常常有效，拓展了知识库中的知识相关性，它可用来建造关于情况和数据的有趣和新颖的解释，猜测属性值，提示可能行得通的方法等。

Agent 理论要建立在整体论、系统论、超协调、耗散结构、社会学、行为学、心理学、协同学和思维科学等众多交叉学科基础上。生物学证明，构成人类大脑智慧物质基础的核心是神经细胞，而神经细胞不过是由我们所熟悉的原子组成，然而我们却始终不明白智慧的奥秘，原因是智能是一种整体特性的表现形式，是大量组成成分——神经细胞的综合能力的表现。心理学实验表明，人的智能具有强烈的时效性、灰度性和层次性，因此，智能模型应是一种多维多层、动态互联的综合网络。思维科学也证实，人的思维是形象思维、抽象思维和灵感思维的综合。耗散结构理论认为，非线形系统具有开放性（通信、协商、合作等），也正是“智能”集成能力所必备的特性。

(3) BDI 模型下的软件 Agent 编程

Agent 的 BDI 理论最初是作为一种分布式智能的模型提出的，当前的研究主要有两个方面的动力，一是控制分布计算的复杂性，二是克服人机界面的局限。

随着任务复杂度的增加，当前流行的直接操作界面的优势已逐渐消失，人们试图用 Agent 技术实现一种间接管理的风格，无需用户显式地告诉计算机的具体行为，而只需给出大体的指导即可。

根据 Agent 的 BDI 理论模型，面向 Agent 编程的具体过程为：

问题综合信念 需求满足愿望规划意图决策行为（软件 Agent）

为了与综合法编程比较，我们给出分析法编程的过程：

问题分析算法描述流程图编码程序执行结果

总之，常规程序是用分析法编写程序，软件 Agent 是用综合法编写程序。

5.6 基于软件 Agent 的分布式体系结构 ADA

为了能够完成分布式计算任务，本节给出了基于 CORBA 的 Agent 的分布式体系结构 ADA（Agent－based Distributed Architecture）。ADA 的主要目的是从动态的、开放的、异构的网络中获取信息，在 CORBA 机制的支持下，完成分布式计算任务。

目前有关 CORBA 和移动 Agent 的使用情况大致分为以下两种：一种是平行使用移动 Agent 和 CORBA，即将两者用于不同的场合，不考虑两者的相互关系。另一种是同时使用 Agent（特别是移动 Agent）和 CORBA，并且注重两者的通信交互。在 CORBA/IIOP 的强大技术和市场优势下，这是目前很多移动 Agent 系统开发和研究者工作的重点之一。这些工作主要集中在移动 Agent 与 CORBA 对象互操作的研究和实践上，它们在一定程度上使得移动 Agent 技术和 CORBA 弥补了对方的不足，但 CORBA 和移动 Agent 仍然是分离的两种技术，分别具有各自的生命周期和对象模型。

Agent 技术，特别是多 Agent 技术，为分布开放系统的分析、设计和实现提供了一种崭新的方法，被誉为是“软件开发的又一重大突破”，Agent 技术已经被广泛应用到各个领域。尤其是随着 Internet/WWW 技术的日益发展和其应用的不断深入，Agent 技术在 Internet 上的应用以及有关移动 Agent 的研究变得愈加活跃。将移动 Agent 的自主移动特性移植到 CORBA 对象，会极大地提高 CORBA 对象的可移动性和灵活性。

5.6.1　ADA 的体系结构

ADA 是由许多相互合作的静态 Agent 以及移动 Agent 组成，Agent 之间采用 KQML/KIF 进行通信。Agent、资源、机器和用户可以根据地理位置、应用范围、组织关系等组成一个逻辑单元，称作领域。在 ADA 中，任何两个 Agent 之间，不管它们是否在同一个领域中，只要它们能够通过代理 Agent 找到对方的话，它们之间都是可以相互通信的。ADA 的体系结构如图 5-5 所示。

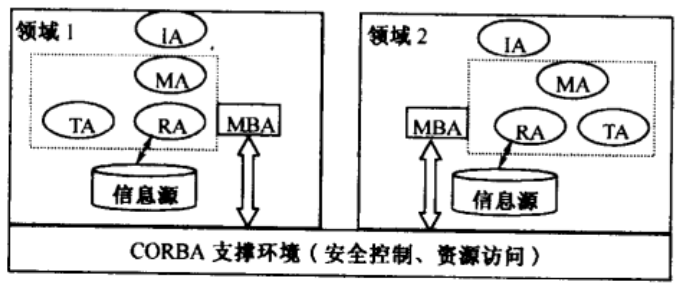


图 5-5　软件 Agent 的分布式体系结构

RA：资源 Agent；TA：任务 Agent；MBA：多代理 Agent；

IA：接口 Agent；MA：管理 Agent

ADA 的核心由一组 Agent 组成，加之基于 CORBA 构建，所以具有很好的伸缩性和可移植性。在 ADA 中采用 KQML 和 KIF（Knowledge Interchange Format）分别作为 Agent 的通信语言和通信内容的格式。KQML 是当前使用极为广

泛的多 Agent 通信协议，它是美国 ARPA 的知识共享计划（Knowledge Sharing Effort）的一部分。KQML 既定义了消息格式也定义了消息传输体制，它为多 Agent 系统中的通信和合作提供了一般框架。KQML 定义了一种 Agent 之间传递信息的标准语法以及一些动作。KIF 是 Agent 通信语言采用很广的一种信息内容格式，它具有很强的表达能力和简单的语法。在 ADA 中通过 Java 语言的多线程技术可以支持 Agent 之间的并发 KQML 对话，并异步执行子任务。

使用消息传递可以为实现复杂的合作策略提供一个灵活的基础。在基于消息的概念中，Agent 之间相互交换的消息可以使用协议来建立通信和合作机制。使用消息实现合作策略必须清楚两点：

① 定义通信协议，准确说明通信过程、消息格式和选择的通信语言。

② 所有相关 Agent 必须知道通信语言的语义，这一点对知识交换是非常重要的。

5.6.2 ADA 中的多代理技术

在 ADA 系统中，Agent 之间在合作过程中是通过代理 Agent 寻找合作伙伴的。代理 Agent 是具有特殊能力的 Agent，它维护着一个有关其他 Agent 的资源内容和任务完成能力的知识库，它可以比较请求 Agent 的要求内容和服务 Agent 的服务内容。例如，当一个被称作请求 Agent 的 Agent 需要寻找一个能完成一项处理任务的 Agent，但它自己并不知道哪个 Agent 更适合时，请求 Agent 便询问代理 Agent 要求提供一个或多个有能力帮助完成这个任务的 Agent。向代理 Agent进行查询，并不表示请求 Agent 附带有许诺。一旦请求 Agent 获得服务 Agent的名字，它也可以决定不向服务 Agent 送去相应的问题。同样情况对服务 Agent 则不同，当它向代理 Agent 宣传了它的服务的有效性后，它也同时对相应的任务作出承诺。在一个系统中如果只有一个代理 Agent，它有可能会带来单点故障和扩充能力的限制。所以，在 ADA 中采用多代理 Agent 技术。多代理的主要目的是使系统更具有健壮性、灵活性和可伸缩性。这种结构的好处是任何代理 Agent 可以在系统中任何其他的代理 Agent 上自由地登记或注销，因此它更具有可升级性，更重要的是这种拓扑结构确保了没有单点故障。在多代理 Agent 环境中，一个代理 Agent 在它的知识库中不仅仅具有其他 Agent 的信息，也具有它所知道的其他代理 Agent 的信息。

5.6.3 ADA 的接口模型

从软件体系结构的角度看，ADA 可划分为三层，并有三个接口（图 5-6）。最底层是 Java 虚拟机 JVM（Java Virsual Machine），它是 Agent-CORBA 对象的运行环境，并通过 JDK API 的形式为程序员提供基本编程接口；中间层是

Agent-CORBA支持环境，它是整个 Agent-CORBA 的核心，并向上为 agent-CORBA应用程序提供编程接口 Agent-CORBA API；最高层是用户 GUI。

图 5-6　ADA 的实现体系结构

ADA 中的 Agent 可以通过 ACL（Agent Communication Language）与其他移动 Agent 和静态 Agent 通信或访问移动Agent服务环境所提供的服务。移动Agent管理与控制模块是移动Agent服务环境的中心部件，它将有关 Agent-CORBA 对象正常运行所需的各项服务正确分配给相应的模块，并协调各个模块的正常运行。该模块的另一个重要功能是实施 Agent-CORBA 对象的约束机制，并根据约束条件控制各个模块的运行。

整个 Agent-CORBA 支持环境需要进行安全性控制和资源访问控制。安全控制模块主要用于实现主机的安全性策略，如进行数字签名验证、对象代码的解密/解压缩等工作。资源访问控制模块则控制 Agent-CORBA 对象对本地资源的访问，并进行付费检查等。

Agent-CORBA 对象具有移动 Agent 的特性，所以它可以移动到其他主机的服务环境，也可以和本地的其他 Agent 进行通信。

Agent-CORBA 是具有移动 Agent 特性的 CORBA 对象，所以它使用 ORB 的命名服务。需要说明的是，为了确保在服务器方 Agent-CORBA 对象移动的情况下保证对象命名服务的透明和持久性，即确保客户方 Agent-CORBA 对象仍能找到移动后的服务器对象，要求服务器方 Agent-CORBA 对象在移动前要从 CORBA 命名服务器中注销，在移动到新的主机后再注册。

基于 CORBA 的多 Agent 分布框架模型 ADA，是大规模软件构架和开发的理想模型。

5.7　基于 Agent 技术的应用开发

基于 Agent 的开发方法，要求构造一些复杂的、自我包含的构件，并且能和其他一些独立开发的类似构件灵活交互。交互作用不再严格通过预定义的接口进行，而是通过参与者之间的协商、劝辩、承诺、同意等行为来实现。这意味着系

统的整体性质和行为不能预先在程序中固定，而是在运行时通过各 Agent 的行为和交互作用动态体现，面向 Agent 的系统正在解决传统方法所不能解决的问题。

5.7.1 面向 Agent 的系统特点

面向 Agent 的系统主要有如下两方面的特点：

① 开放性。在 Internet 这样的环境中，基于 Agent 的方法之所以这样的重要，在于用户不能预知与其交互的各种资源。这要求 Agent 及它们之间的接口必须具有灵活性和坚定性。但是在设计 Agent 时很少能预测到将来可能发生的行为，因此 Agent 必须具有高超的社交能力，使他人同意自己观点的说服能力，有助于达成一致的协商能力，并在适当时期主动产生新的目标。

② 复杂性。面向 Agent 系统比起传统软件工程中经典的“分而治之”策略来说，其优越性在于，Agent 能够以灵活的、上下文有关的方式（因此需要社交能力和主动性），而不是一些预先严格规定的接口函数，与外界进行交互作用。环境的不可预测性要求 Agent 既能对环境的变化作出反映，又能根据目标主动规划自己的行为。

此外，面向 Agent 的系统还具有通用性、模块性、重用性、扩展性、移植性、智能性、灵活性等特点，在此不再一一叙述。

5.7.2 面向 Agent 的应用开发步骤

面向 Agent 技术的应用开发可以按照下述步骤进行：

① 分析系统的特点，选择合适的实现技术。在进行应用开发时，首先应根据实际情况决定应采取何种技术实现。一般而言，当应用需要具有跨平台、跨网络、跨地域甚至跨行业的互操作性以及较高的个人化、智能化时可以考虑采用 Agent 技术。这里并不排除以多种技术实现应用系统的可能性。在确定以 Agent 技术实现应用系统后，应当具体分析应用所涉及的各个对象，决定哪些采用 Agent实现，哪些采用其他方法实现。

② Agent 的功能设计。确定系统采用 Agent 技术实现部分的数据和功能。Agent间明确分工后，应当根据各自功能确定内部数据。此外还要考虑 Agent 的种类：移动 Agent 还是静态 Agent。此外，移动 Agent 的内部数据应尽可能的少，以减少移动带来的网络负担。

③ Agent 接口的设计。Agent 接口的设计非常关键，它往往影响系统的性能。这时既要考虑 Agent 间的交互方式，又要考虑 Agent 与非 Agent 部分的交互方式。

④ Agent 的详细设计和实例化。首先要了解目前已有的 Agent 平台能做什么，不能做什么，它们各有什么优缺点。然后根据系统的需要选择合适的代理平

台。接着进行详细设计和具体的编码、调试工作，从而实现整个系统。

⑤ Agent 的运行与维护。运行维护基于 Agent 技术的应用系统，及时发现和解决实际运行过程中遇到的问题。

总之，任何系统的设计都要综合应用多方面的因素，根据实际情况而定。

5.7.3　面向 Agent 技术开发中存在的问题

面向软件 Agent 技术的应用开发虽然可以解决以前许多难以解决的问题。但是，Agent 技术本身固有的特性限制了人们对它的信任，加之 Agent 可以自由作出决策，使得 Agent 之间的依赖关系难以得到有效的管理，同时也影响到用户的自治性。对 Agent 系统的开发来说，面临着下述问题：

① 缺乏有关基于 Agent 的软件系统开发方法学，这使 Agent 的开发无规范可言。

② 对于不同的基于 Agent 软件系统的设计方案缺乏评估、比较的标准，这使得 Agent 系统的开发设计人员对于在何种情况适合采用何种机制几乎无章可循。

③ 与可重用技术和分布式对象技术的结合不够紧密。为了解决阻碍软件 Agent发展的有关技术问题，一方面需要自底向上建设适合软件 Agent 开发的基础设施，为 Agent 系统的构造提供有利支持；另一方面，更需要自顶向下对 Agent系统的构造理论、方法和体系结构进行研究。

④ 与用户自治性之间存在矛盾。Agent 的自主性和能动性是 Agent 的基本属性。其中 Agent 的自主性是指 Agent 能够不依赖于外界的直接干预而独立运行，对自身的行为和内部状态有某种控制权；Agent 的能动性是指 Agent 能够能动地采取有目的行动而不仅仅是简单地响应环境。由于 Agent 具有自主性和能动性，Agent 在使用过程中不可避免地会影响用户的自治性。当用户发现使用 Agent 无法确保自己的自治性，用户会对 Agent 不信任而不愿意使用 Agent。用户自治性是一个独立的特征。如果软件设计人员在设计 Agent 时没有考虑这些问题，那么用户在使用 Agent 时其自治性就会受到损害。所以用户自治性已经成为设计 Agent系统时要考虑的重要的因素。

参 考 文 献

程显毅，董宏斌. 2000. 设计 Agent 系统应注意的问题. 计算机工程与应用，36(11)：64～65

程显毅，杨健，盛文. 2001. 软件 Agent 是一个计算实体. 计算机工程与应用，22(1)：41～44

董军等. 2000. 路由选择的多 Agent 模型. 计算机学报，23(2)：221～225

樊玮，朱军. 1999. 软件 Agent 技术的研究. 航空计算技术，29(4)：34～27

沈宁川，龙翔，聊鸿斌. 1997. 一个维护知识库的软件 Agent 互操作系统. 软件学报，8(1)

孙玉冰，林作铨.2000. 软件 Agent.计算技术与自动化，19(1)：75～77

陶青萍，蔡庆生.2001. 一种新的软件 Agent 设计的评价标准：用户自治性.小型微型计算机系统，21(2)：147～149

夏幼明，徐天伟，张春霞等.1999. 软件 Agent 的初步研究.云南师范大学学报，19 (3)：8～11

夏满民，李怀诚.2002. 智能软件 Agent 在分布式信息管理中的应用.南京航空航天大学学报，32 (5)：591～597

赵龙文，侯义斌.2000.Agent 的概念模型及其应用技术.计算机工程与科学，22(6)：75～79

Gao Ji, et al. 1999. ABFSC: An agent-based framework for software composition. Chinese Journal of Computers, 22(10): 1050～1058 (in Chinese)

Genesereth M, et al. 1992. Knowledge Interchange Format Version 3.0 Reference Manual Technical Report. Computer Science Department, Stanford University, California

Genesereth M R, Ketchpel S P. 1994. Software agent. Communication of the ACM, 37(7): 48～53

Wooldridge M. 1997. Agent-based software engineering. IEEE Transactions on Software Engineering, 144(1): 26～37

Wooldridge M, Jennings N R. 1995. Intelligent agents: theory and practice. The Knowledge Engineering Review, 10(2): 115～152

第 6 章　大规模软件构架中的集成技术

互操作是指客户端和服务端分别基于两种技术之一，当处于一个技术平台中的客户端向另一平台中的服务端发出请求时，通过一定的转换和编码，使服务器能够接受客户请求，执行相应操作并将处理结果返回给客户。

在基于互操作技术的基础上，可以实现软件的集成。

6.1　多数据库集成

实现信息集成的主要途径之一是建立多数据库集成系统。多数据库集成系统为用户提供单一类型的数据定义和操作语言，允许访问多个独立的数据库。这是通过对成员数据库的相关部分进行转换和集成，为用户建立统一的集成模式和接口而完成的。一个分布式计算机信息系统的异构性可划分为 3 个层次。最底层是平台层，如不同的硬件、操作系统或通信协议；中间层是系统层，如不同种类的数据库管理系统，甚至文件系统，它们基于不同的数据类型，提供不同的语言；最上层是语义层，由于不同的数据库或文件是独立设计的，不同系统中的数据语义之间存在着冲突。

在当今社会里，数据库的使用无处不在，但就信息源而论，多种数据库同时存在也是客观现实。数据库发展经历了层次、网状、关系数据库系统，以及近几年发展起来的对象——关系和完全面向对象的数据库系统。对一个大的企业，各部门使用不同的数据库系统是经常出现的情况。即使一个经过严密规划的企业部门，因时间推移，人事变迁，以及数据库技术的发展和数据库市场的变化，都有可能造成异构数据库并存的局面。另一方面，随着数据库技术的发展，各数据库公司自身也不断推出新的数据库版本。随着应用的日趋复杂化，数据库已不仅仅局限于简单的信息存储和处理，它成为网络技术、面向对象技术、分布式技术和并行处理等多种技术的焦点。因此，在现代生活中，异构分布计算环境下多数据库系统联合和集成使用是非常迫切并会长期存在的问题。

对于普通用户而言，希望屏蔽掉各种层次的异构特性，他们并不知道各物理数据库系统的分布和结构组成，也不必自己进行数据转换和结果汇总，只需通过一个简便的全局查询得到一个综合的结果。这也是多数据库系统集成研究的主要内容。它的研究目标是将地理上分布的多个异构数据库，在尽可能少的影响本地自治性的基础上，构造用户所需要的透明性的全局数据库，以支持各数据库的全

局应用和各异构数据库之间信息的灵活交换和共享。

由此可见，异构分布计算环境下多数据库系统的集成是指在计算机网络环境中，用户不仅能够实现对多个异种数据库的透明访问，而且不同数据库之间可以相互协作，即各种数据库系统上的应用系统在集成系统的支持下，不关心底层网络通信问题和不同数据库系统的模式匹配问题，也不必关心操作语言的转换和存取路径等细节，即这种集成具有异构透明性、分布透明性和协作透明性等3种透明性。

异构环境下多数据库系统的集成所面临的问题有以下3大类：

① 系统问题。包括两个方面：一方面是多数据库系统应提供哪些方面的功能，即在所提供的统一接口里面应包括哪些内容，以及该接口怎样使用等；另一方面是所设计的多数据库系统必须是可实施的、可扩充的、可升级的、可移植的，同时要能实现与其他系统的互连和互操作。

② 语义问题。多数据库的语义集成是非常重要的。语义异构是指相同和相关数据的不同理解、使用和表达。由于数据语义和行为语义表达了多数据库之间的依赖关系，多数据库系统的集成模型必须能够描述原数据、中间数据、集成数据、局部数据库系统和应用系统的语义，多数据库系统的模式转换和集成必须是带有语义的。

③ 集成问题。集成问题涉及数据集成、控制集成和视图集成等3个方面的内容。其中，数据集成主要解决同义词、同名异义词、取值范围冲突、数据丢失、值冲突、语义冲突、结构冲突、不同类型对象之间的映射与转换等问题；控制集成是指在集成的过程中，数据对象和应用对象的行为语义必须予以保留或进行相应的转换；视图集成是指在集成环境里不同层次的抽象和集成，这种视图集成要求采用面向对象技术，以保证在集成的过程中数据对象对应用系统保留合适的状态和行为语义。

以上3类问题的核心点是解决异构分布计算环境下数据库的互操作问题。关于数据库互操作问题的研究，国外学术界和产业界主要是沿着以下两个方面在努力：一是发展数据库的有关标准；二是承认各厂商出品数据库的千差万别，因势利导，建立数据库互操作支撑件。发展有关标准无疑可以使异种数据库互操作的实现基础更好，起点更高；但标准的建立难度大，时间长，且常常无法解决当务之急。因此，在建立标准尚需时日和符合标准的产品尚不完善的情况下，人们常把解决数据库互操作的注意力放在构造数据库互操作模型上。

6.1.1 基于CORBA的多数据库集成的内容

实现数据库互操作支撑模型必须解决好3个问题：透明性、响应性和安全性问题。其中透明性问题又是首要的核心问题。现有的数据库互操作支撑模型往往

采用传统的网关方式来构造数据库互操作环境，尽管它们在一定程度上解决了数据库的类型透明性和位置透明性问题，但由于体系结构的局限，在响应性能和安全性方面常存在缺陷。

基于 CORBA 规范，构造数据库互操作支撑模型来实现数据库的互操作，使之成为 CORBA 的一种应用，不仅可以解决透明性和安全性问题；同时，由于在体系结构上的突破，使其响应性能也能得到改善。由此可见，CORBA 解决了平台的异构性问题，但也为解决数据库系统的异构性提供了基础结构。基于 CORBA 软件总线的多数据库集成技术主要包括以下 4 项内容：

① 面向对象集成模型及查询语言的实现技术，包括解决查询处理、查询优化以及从全局执行语言到集成对象方法的联编技术。

② 面向对象的全局事务管理和执行技术。CORBA 软件总线系统提供了对象应用，完成作为对象的事务创建、调度、执行、提交和取消，保证全局事务的原子性、一致性、隔离性和持久性。

③ 面向对象的集成技术。包括局部数据的对象化表示、从局部模式到标准模式的转换、语义捕捉和精练、语义冲突的解决、集成字典管理和维护等。

④ 基于软件总线的集成结构和方法学。软件总线具有支持对象的实现透明性和分布的透明性等特点。在此结构基础上，有关多数据库集成体系结构、功能层次划分、集成模式结构以及有关对 OO 开发的支持环境等，都是新的研究课题。

由于 CORBA 已成为具有一定影响的国际标准，实现基于 CORBA 的多数据库集成是大规模软件构架中异种数据库集成的有效途径。

6.1.2　基于 CORBA 的多数据库集成实现策略

CORBA 的分布体系结构将一个分布式异构系统作为相互作用对象的集合，即将分布式系统中各个成员系统的资源模型化为对象，而成员系统提供的服务被模型化为对象方法，用这些方法组成对象接口。这样，在每个成员系统上可以定义一个服务接口，成员系统为这些服务提供实现。客户通过以公共语言表示的请求与异构系统进行交互，对象请求代理机制负责转换客户请求到可用服务、传递请求到适应系统，提供公共语言表示的应答给客户。

实现基于 CORBA 的多数据库集成系统的基础是，将参加集成的数据库通过对象包装注册到 ORB 总线上。在注册各种数据库到 ORB 时，有如下一些关键的实现策略：

① 对象粒度。CORBA 体系结构中，对象可按各种粒度大小来定义。

② 调用方式。CORBA 允许两种调用方式：动态方式和存根方式。

③ 客户请求到服务器的映射。CORBA 提供了一个接口对应一个实现、一个

接口对应多个实现之一、一个接口对应多个实现等3种方式。

④ 对象生命周期。

⑤ 激活策略。是指如何在服务器上启动每个对象的实现。可以使用共享式、非共享式、单个方式和持久式4种激活策略。

⑥ ORB联编策略。ORB提供了静态、自动、动态3种解决对象引用的策略，即将一个请求联编到某个实现上。

基于CORBA实现的多数据库集成系统在处理已有应用和系统方面有其独到的优势，用户无需抛弃已有的应用，只要在已有的应用之上实现一个接口，就能集成到CORBA系统中，保护了企业已有的投资。未来基于CORBA的多数据库系统应用应该是一个跨越不同地理位置、穿越不同网络环境、屏蔽实现细节、实现透明传输、集成不同用户特长的、面向对象的、开放的分布式集成系统。总的来说，基于CORBA的数据库集成和访问具有如下的优点：

① 屏蔽了数据层数据结构和表示方式的异构问题；

② 代码和功能模块具有更强的可重用性；

③ 容易进行各种未来数据库系统的集成。

6.1.3 基于CORBA的多数据库集成结构和访问途径

1. 实现结构

基于CORBA的多数据库集成结构的主要构件如下：

① 视图支持。实现视图的多层抽象与封装，提供视图接口，满足不同层次的应用需求。

② 客户接口。通过两种方式（静态调用和动态调用）实现客户服务器对象的请求。

③ 对象实施。由数据和代码构成，数据描述对象的结构和状态，代码描述对象的操作行为。

④ 对象适配器。其作用体现在提供了调用ORB核心服务的接口，可以实现其他对象模型与CORBA基本对象模型之间的映射。

⑤ ORB接口存储。支持动态调用，存储有关对象运行时期的IDL信息，实现应用请求与服务器对象的联编。

⑥ ORB Core。是客户和对象实施之间的一个代理，其功能包括对象的定位、消息的传递、方法联编等。

⑦ ORB通信服务。实现ORB与底层互连网络的集成。

2. 访问途径

对基于CORBA的集成数据库的访问有两种途径，一是在访问数据库的过程

中，可以将 CORBA 对象作为关系数据库的前端，根据数据库的结构构造相应的 CORBA 对象，使用 IDL 语言描述其数据结构，客户端使用 IDL 的 Stub 代理与 CORBA 对象进行通信，而由 CORBA 对象实现对数据库中数据的存取、查询和更新。将 CORBA 组件作为数据库访问的中间件，这种方法较为直观，开发起来较为容易，缺点是缺乏通用性，要针对不同的数据库的结构来编写相应的对象。另一种途径是将 CORBA 对象与 Java 语言相结合，Java 作为数据库的前端提供了访问一致性的接口 JDBC－API。使用 JDBC 技术实现对 CORBA 组件的封装，可以不关心数据库的逻辑结构，其逻辑结构由客户端进行处理，数据的存储则由数据库管理系统负责。在这种情况下，由客户端指定连接的参数，而由 CORBA 对象来完成对不同后台数据库的连接。

6.1.4　基于 COM＋与 ASP 技术的多数据库集成

1．基于 ADO 的数据库访问技术

ADO 对象模型定义了一组可编程的自动化对象，可用于 Visual Basic、Visual C++、Java 以及其他各种支持自动化特性的脚本语言。ADO 最早被用于微软的 IIS 中访问数据库的接口，与一般的数据库接口相比，ADO 可更好地用于网络环境，通过优化技术，它尽可能地降低网络流量；ADO 的另一个特性是使用简单，不仅因为它是一个面向高级用户的数据库接口，更因为它使用了一组简化的接口用以处理各种数据源。这两个特性使得 ADO 必将取代 RDO 和 DAO，成为最终的应用层数据接口标准。

从图 6-1 可以看出，ADO 实际上是 OLE-DB 的应用层接口，这种结构也为一致的数据访问接口提供了很好的扩展性，而不再局限于特定的数据源，因此，ADO 可以处理各种 OLE－DB 支持的数据源。

在 ADO 应用中，主体对象只有 3 个：Connection、Command 和 Recordset，其他 4 个集合对象 Errors、Properties、Parameters 和 Fields 分别对应 Error、Property、Parameter 和 Field 对象，整个 ADO 对象模型由这些对象组成。

与数据库的所有通信都需要打开一个连接来进行，这是通过 ADO 的连接对象来完成的。在对一个数据库进行数据的插入和读取之前，必须打开与这个数据库的连接。Command 对象是针对数据源执行特定命令的一种定义，使用 Command 对象可以查询一个数据库并将记录返回到一个 Recordset 对象中，或者执行一个批量操作，或者维护数据库结构。Recordset 对象是 ADO 对象模型中最为灵活的一个，利用它，可以很方便地操作数据库中的记录。一个 Recordset 对象代表一个数据库表中的一整组记录或一次命令执行的结果。在任何时候 Recordset 对象只能指向当前记录所在记录集中的一条记录。

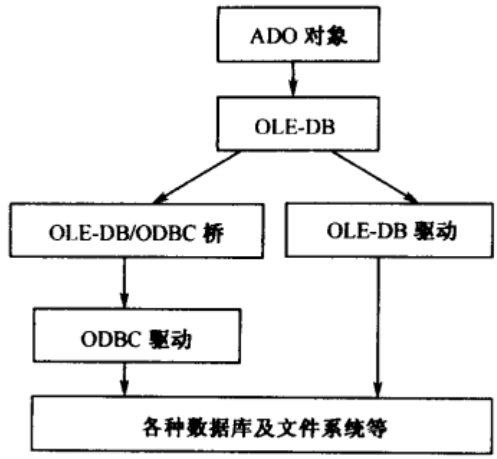


图 6-1 ADO 与数据库 API 之间的层次关系

一个典型的 ADO 应用使用 Connection 对象建立与数据源的连接，然后用一个 Command 对象给出对数据库操作的命令，比如查询或者更新数据等，而 Recordset 用于对结果集数据进行维护或者浏览等操作。Command 命令所使用的命令语言与底层所对应的 OLE-DB 数据源有关，不同的数据源可以使用不同的命令语言，对于关系型数据库，通常使用 SQL 作为命令语言。

在 Connection、Command 和 Recordset 3 个对象中，Command 对象是个可选对象，它是否有效取决于 OLE-DB 数据提供者是否实现了 ICommand 接口。由于 OLE-DB 可提供关系型数据源也可以提供非关系型数据源，所以在非关系型数据源上使用传统的 SQL 命令查询数据有可能无效，甚至 Command 命令对象也不能使用。ADO 的对象模型如图 6-2 所示。

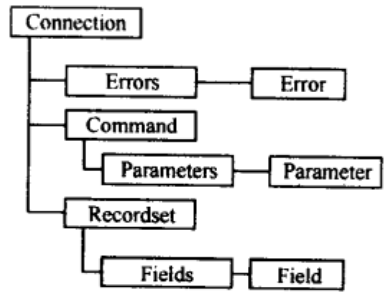


图 6-2 ADO 对象家族树

2. 数据库访问中间件模型

为了实现数据库访问的透明性和一致性，在此给出利用 COM + 和 ADO 技

术构造一个数据库访问的中间件模型，如图 6-3 所示。该模型符合 COM + 规范的对象，对象的实现可以采用 Visual Basic 或C++ 语言来编写，数据库访问部分使用 ADO。客户应用程序可以通过 Form 对象或浏览器对象实现对数据库的访问。对客户应用程序提供一致的界面，使其不必关心后台数据库的具体连接和通信。即该模型是按 COM + 规范进行封装 ADO 对象的实现。

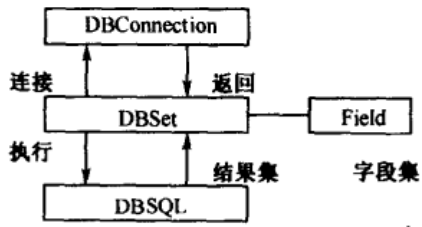


图 6-3　数据库访问中间件模型

在中间件模型中，实现了 DBConnection、DBSQL 和 DBSet 接口，其中 DBConnection 接口用于数据库的连接，支持数据库的事务（transaction）处理，它使用 ADO 对象中的 Connection 方法建立与数据库的连接，对应 ADO 的 Connection 接口；DBSet 对应 ADO 的 RecordSet 接口，它是用 SetNew 语句创建的一个新的 Recordset 对象，它提供对所产生的数据表进行访问。DBSQL 是向数据库发送的 SQL 查询或存储过程的调用代码，包括 SQL 修改、插入、删除或查询语句，它使用已有的连接对象初始化 Recordset 的 ActiveConnection 属性并且指定 Source、CursorType 和 LockType 属性值。

基于 COM + 构造数据库访问中间件，与基于 COM/DCOM 的 ADO 一样，一方面它能实现分布式的组件对象和实现多层的应用程序结构，另一方面由于 COM + 继承了 COM、DCOM 和 MTS 的许多特性，同时也新增了一些服务，比如负载平衡、内存数据库、事件模型、队列服务等。COM + 新增的服务为 COM + 应用提供了很强的功能，建立在 COM + 基础上的应用程序可以直接利用这些服务而获得良好的应用特性，从而可以构建功能更为广泛的 COM + 构件。

3．基于 ASP 的多源数据库集成

ASP 是微软 IIS3.0 以后新增的功能，具有以下优点：

① ASP 将脚本制作语言 VBScript、JavaScript 和 HTML 结合起来形成 Web 页面，克服了 Web 页面的局限性，使 Web 页面从静态变为动态，是一个飞跃。

② ASP 的工作机制消除了对于 Web 浏览器的依赖性，即客户方的浏览器可以采用任何厂商的产品。浏览器向 IIS 请求 ASP 文档，IIS 解释其中的脚本语言，并产生标准的 HTML，传送给客户端的浏览器。由此可见，所有的处理都

是在服务器上进行的。

③ 代码的安全性。基于 ASP 页面的访问者所能看到的只是服务器的处理结果。

④ ASP 提供了许多开发者可利用的内置组件对象，可管理变量到提交表单的所有内容。

⑤ ASP 允许用户开发自己的 COM 组件来封装事务逻辑。组件是可重用的，当多个页面进行相同的事物处理时，调用同一组件即可。但事务逻辑发生变化时，无需改变页面脚本，只需调用相应的组件即可适应事务逻辑的改变。

在 ASP 中使用组件是非常容易的，只要将 HTML、脚本语言和组件等结合到一起，才能建立一动态的、功能强大的 Web 应用系统。所以，在多源数据库应用程序设计中，首先是构建服务器中基于 COM + 的构件，构件对象通过 OLD - DB 或 ADO 的标准接口访问数据库，COM + 实现对构件对象的注册和管理；其次是利用 ASP 技术将多个数据库控件（control）嵌入到 ASP 页面中，浏览器通过这些不同的控件完成对多源数据库的访问。不过，这种多源集成方案，对多源数据库一致性要求不强的应用非常有效。

6.2 CORBA 与 OLE/COM 的互操作和集成

CORBA 和 OLE/COM 是目前分布对象系统中的两大主流，它们有各自的特点和用户群。就环境而言，CORBA 主要面向异构环境下的分布式应用；而 COM 主要面向 Windows 平台，更适应于 Windows 平台上的分布式应用。从目前来看，两者会在相当长的时间内共存。

由于分布式对象系统及应用的开放性，两者之间有着内在的必然联系。一方面，使用 Visual Basic、Visual C ++ 和 PowerBuilder 等编程语言开发的客户应用需要访问 CORBA 对象，而目前 CORBA 不支持 OMG IDL 到 Visual Basic 等语言的映射，因此以这些语言开发的客户应用需要通过 COM 来访问 CORBA 对象；另一方面，Windows 平台有许多基于 COM 开发的应用程序，对于 CORBA 而言，可以将它们看作已有系统，通过包装集成，可为 CORBA 应用提供服务。

随着分布式计算对于企业应用拥有越来越重要的作用，各种标准之争越来越火热，最为明显的就是 DCOM 与 CORBA。故目前企业开发的应用最好能兼顾这两种标准，实现 DCOM 与 CORBA 的互操作和集成。目前比较流行COM-CORBA 桥或利用 COM 和 CORBA 兼容的 API 技术方案，有一些公司已提供成熟的商业产品，但这种方案会造成缓慢的程序运行效率，也不利于系统的扩展。如果在应用服务器中能直接提供两种标准的支持，才能保证分布式程序的运行效率，特别适应于需要大量数据进行处理的系统相连上。也就是说，在一个应用服务器中同

时提供 DCOM 与 CORBA 接口，同时支持 DCOM 客户与 CORBA 客户。由此可见，研究和实现分布式对象系统 CORBA 和 OLE/COM 的互操作非常的重要，可以增强分布系统的集成能力，为数字城市的分布式构架奠定基础。然而，由于两者分属于不同的模型对象系统，无论在模型框架还是在通用性、扩展性、重用性、可移植性及实现技术上都存在较大的差异。因而，实现 CORBA 和 OLE/COM的集成存在一定的难度。下面从互操作和集成两个方面进行阐述。

6.2.1　CORBA 与 OLE/COM 的互操作

1．互操作模型的视图映射

在 CORBA 与 OLE/COM 的互操作模型中需要在两个方向提供视图映射，并存在 4 种形式：CORBA 目标的 COM 视图映射、COM 目标的 CORBA 视图映射、CORBA 目标的 Automation 视图映射、Automation 目标的 CORBA 视图映射。

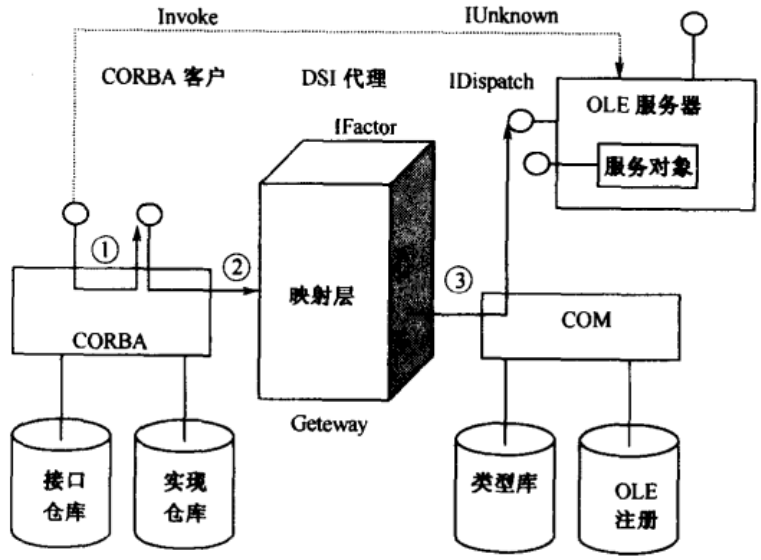


图 6-4　通过动态调用网关 CORBA 调用 OLE 服务

OMG 规范的互操作实际上是一种概念结构，它在物理上有多种方式，主要有通用转换桥和专用转换接口两类。下面给出通用转换桥的思想。

CORBA 与 COM 的互操作包括 CORBA 调用 COM 的服务和 COM 调用 CORBA 的服务。实现中都要用到两个对象系中的类型库、接口仓库和实现仓库，主要用于请求动态构造、类型转换和类型查找等。

图 6-4 通过动态调用网关（一种通用转换桥）实现 CORBA 客户实体对 OLE 对象服务的调用。在 CORBA 端通过一个 DSI 代理对象向网关的映射层发请求，

网关对 COM 来说相当于一个客户实体，它对 CORBA 系来的对象请求经过一定转换后调用目标对象的 IDispach 接口，将请求分发给目标对象的被请求方法。

图 6-5 是通过动态调用网关实现 OLE/COM 客户对象对 CORBA 对象服务的调用。在 OLE/COM 端通过一个代理对象（proxy）向网关的映射层发出请求。网关对 CORBA 来说相当于客户，它对来自 OLE/COM 对象的请求经过一定的转换后分发给目标对象的被请求方法。

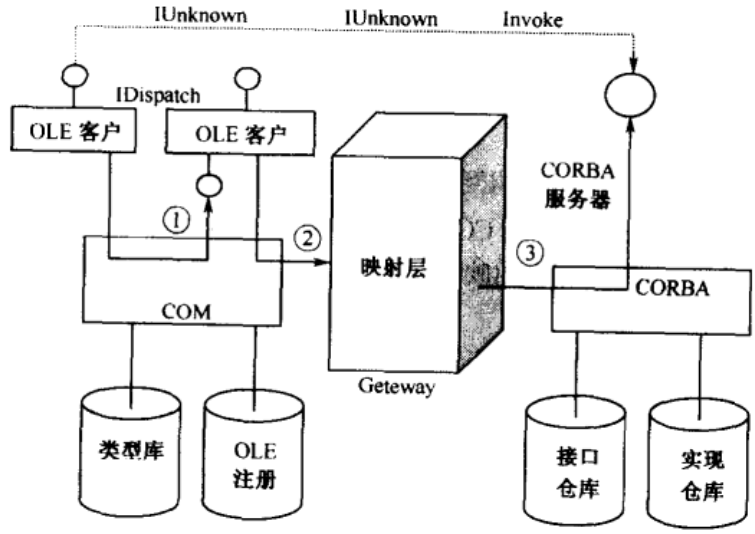


图 6-5 通过动态调用网关 OLE 调用 CORBA 服务

OMG 规范推荐了两种互操作实现策略，通用映射和特定界面的映射。

① 通用映射。所有的界面都由单独的互操作构件动态地进行映射。当目标系统中注册了新的界面和对象时，用户可以直接去访问这些新的对象而无需针对这些界面专门生成互操作代理。这种方法简化了安装和改变的管理，但是性能较低。

② 特定界面映射。在访问目标对象之前，用户需要使用互操作编译器对目标对象的界面描述文件进行编译，生成互操作代理，之后才可以通过该代理访问目标对象。这种方法提高了性能，但用户使用和管理工作较为复杂。

2．映射要点分析

互操作在实现时需要将 B 中的对象映射为 A 中的视图对象，主要包括以下内容。

（1）界面映射

COM 和 CORBA 都使用界面来描述对象的特性。界面映射定义了界面描述中的属性、操作和数据类型等部件的映射规则。

(2) 界面组成映射

COM 和 CORBA 采用不同的方式描述对象的继承关系。CORBA 支持多继承；COM 只支持单一继承，但对象在实现时可以通过聚集的方式包含多个界面，客户可以通过 IUnknown 界面的 QueryInterface 方法来查询并访问其他界面。OMG 规范分别指定了 CORBA 界面的多继承映射为 COM 和 Automation 单继承的映射规则。OMG 规范没有将 COM 中的聚集映射为 CORBA 中的多继承，这是因为 COM 中的聚集是在对象实现时采用的策略。OMG 规范规定将 COM 对象聚集的界面映射为多个独立的 CORBA 界面，并通过一组 CORBA 界面（CORBA::COMposite 和 CORBA::COMposable）管理这些界面，CORBA 客户可以使用映射后的 QueryInterface 方法得到 COM 对象的其他界面。

(3) 对象生命周期映射

CORBA 中客户和服务器对象的生命周期是分开管理的，而在 COM 中则采用引用计数（reference count）的方式管理服务器对象的生命周期，即客户对服务器对象的引用决定了服务器对象的生命周期。OMG 规范认为 CORBA 的生命周期机制优于 COM 的生命周期机制。为了充分利用 CORBA 对象的生命周期机制，OMG 规范规定 CORBA 对象的 COM 视图对象采用 COM 的生命周期，但视图对象在释放自身时不释放它所代表的 CORBA 对象；COM 对象的 CORBA 视图对象采用 CORBA 对象的生命周期机制。OMG 规范还定义了 ICORBAFactory 界面和 SampleFactory 界面，分别用于 COM 客户得到 CORBA 对象的引用和 CORBA 客户得到 COM 对象的引用。

(4) 映射的可逆性

OMG 规范指出有可能需要映射的可逆性，但没有指出具体的可逆性需求。

6.2.2　CORBA 与 OLE/COM 的集成

CORBA 与 LOE/COM 的集成包括两个方面，一是 CORBA 对 OLE/COM 的集成，二是 OLE/COM 对 CORBA 的集成。图 6-6 和图 6-7 分别给出了这两种集成的机制。

CORBA 对 OLE/COM 服务的集成就是把已有的 OLE Automation Server 构件挂接在 CORBA 平台上作服务器，提供 CORBA 客户方应用的访问。这样使 Windows 平台上已有的各种构件在网络环境下，通过 CORBA 实现异种平台上的应用集成，实现原理如图 6-6 所示。

CORBA 连接对象的实现是 OLE Automation Server 和 CORBA 平台进行交互的中间件，它相对 CORBA 平台来说是服务器，而相对于 OLE Automation Server 来说是客户应用方。当 CORBA 客户应用发出的请求在 CORBA 的连接对象解码后，转发给 OLE Automation Server 执行，并与之交换数据，然后再按照 CORBA

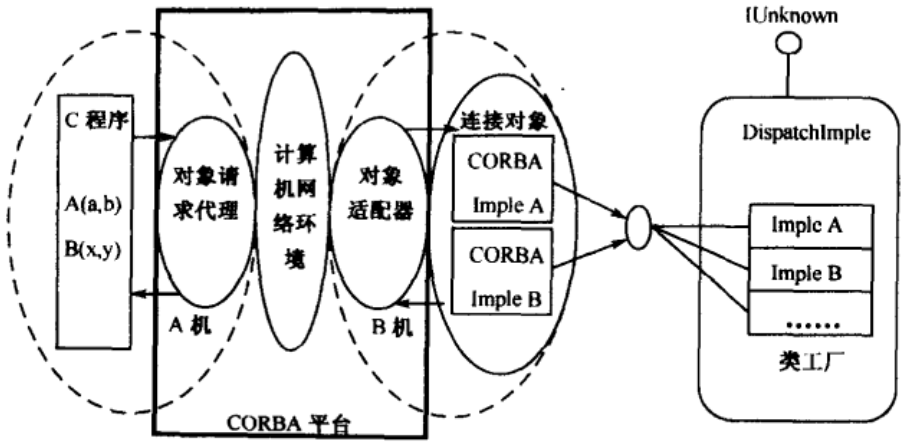


图 6-6 CORBA 对 OLE/COM 服务的集成机理

的标准返回数据。这些操作对 CORBA 客户应用是透明的。CORBA 平台及连接对象屏蔽了 CORBA 客户应用对 OLE Automation Server 服务的调用。

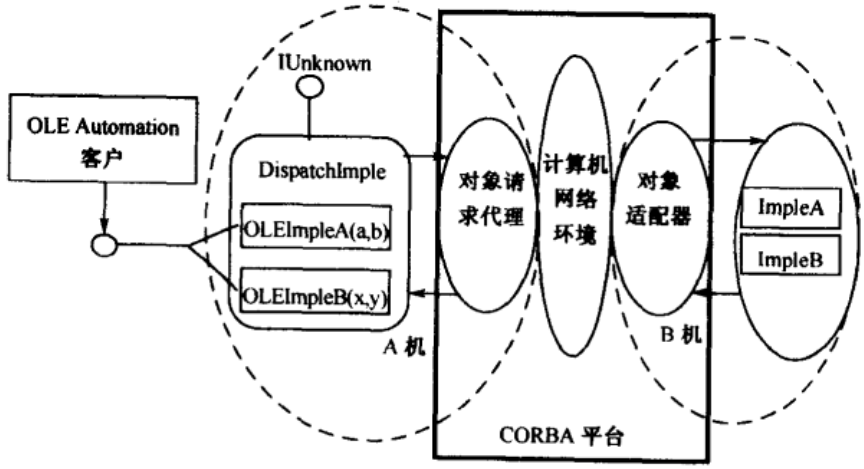


图 6-7 OLE/COM 对 CORBA 服务的集成机理

利用图 6-7 的结构模型可实现 OLE/COM 对 CORBA 服务的集成。CORBA 客户端应用可利用 PowerBuilder、Visual Basic 和 Delphi 等语言针对 OLE Automation Server 进行开发，开发出来的应用通过 OLE Automation Server 访问 CORBA。OLE Automation Server 相对 CORBA 平台是客户应用，而相对于 PowerBuilder, Visual Basic、Delphi 等是服务器。OLE Automation Server 屏蔽了客户应用对 CORBA 服务的调用请求，实现访问的透明性。

根据 CORBA 标准，IDL 编译器只能将接口描述映射为 C、C++，Smalltalk

等语言。这样，只能用上述语言开发客户方应用，这就限制了应用的广泛性，形成了 CORBA 平台上应用的瓶颈。因而通过 CORBA 客户方与 OLE Automation Server 的集成来实现客户方开发应用的广泛性。

6.3　CORBA 与 DCE 的互操作和集成

OMG 提出 CORBA 规范的主要目标是实现软件模块的即插即用。ORB 提供透明的接收对象请求和返回应答机制，根据这种机制，ORB 提供异构分布式环境之间的互操作性。CORBA 是一种分布式对象技术标准，作为一种新兴的中间件技术，在最近几年里也得到了迅速发展，而且因为它是面向对象的技术，提供了良好的移植性和可重用性，可能有更好的发展前景。DCE 的最初目标是屏蔽各种各样的传输协议和计算机软、硬件之间的差异。DCE 主要是面向过程编程模型的，经过多年的发展和应用，DCE 在分布式应用领域中已经得到了广泛的使用，有很多大型的分布式应用系统都是基于 DCE 环境开发的。OSF 在 DCE 1.2 中又扩展了一些面向对象的特性。

CORBA 与 DCE 是两种主要的中间件技术，在技术体系上有很大的相似性，如它们都有接口定义语言（IDL），提供名字服务、进程间通信服务等。但它们各自还有对方没有提供的一些服务，如 DCE 提供远端过程调用服务和面向过程的编程模式，而 CORBA 提供对象生命期服务和面向对象的编程模式，另外 CORBA 还提供 DCE 中没有的动态调用等。就目前而言，它们都有自己的应用领域，并会在较长的时期内持续并存下去。所以，有必要解决它们之间的互操作问题。

6.3.1　CORBA/DCE 互操作的分类

DCE 和 CORBA 的互操作分为如下 4 种类型：

① 客户端是 CORBA，服务端是 DCE。

② 客户端是 DCE，服务端是 CORBA。

③ 客户端是 CORBA，服务端是用 DCE1.2 C++ 扩展实现的 DCE。

④ 客户端是用 DCE1.2 C++ 扩展实现的 DCE，服务端是 CORBA。

不管是上述何种类型，都不可避免地要涉及到下面所说的两个主要问题：

① 不同平台下的接口类型的转换和替代，也就是如何将 DCE 的 IDL 与 CORBA 的 IDL 进行相互等价转换；

② 对象在不同环境中的绑定及互操作。

下面针对 CORBA 作为应用客户端，如何透明地调用 DCE 服务端的问题，根据参考文献（曾飞鹏等，1999）中提供的“桥接”技术，对 CORBA 和 DCE

互操作的实现予以描述。其他情况与此非常相似，在此不再重述。

6.3.2 CORBA/DCE 互操作的实现

1. 桥对象

桥是一个处理过程，它接受客户端发来的请求，经过转化将其传送到服务端，还能将服务端处理的结果返回给发送请求的客户。由此可见，桥实际上就是客户与服务之间的代理。

图 6-8 所示为桥按照时间序的一般操作过程，它所描述的操作步骤的含义如下：

① 桥接收来自客户端发来的请求；
② 桥将参数转化为等价的服务器环境中的参数；
③ 桥调用客户所请求的操作；
④ 服务端执行该操作并将结果返回给桥；
⑤ 桥将输出结果和参数转换为客户端可识别的类型；
⑥ 桥将输出结果和参数返回给客户。

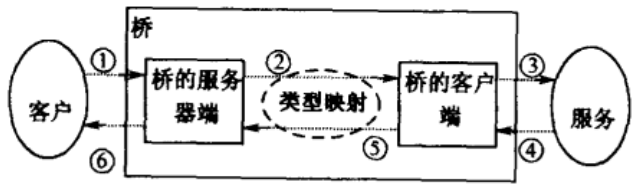


图 6-8 桥的一般操作过程

2. 桥的类型

桥既是服务器又是客户端。对于客户它是服务器，而对于服务它却是客户端。桥含有一个服务端接口（SSI）和一个客户端接口（CSI），分别用来接受客户请求和向服务器发送请求。SSI 和 CSI 分别是用客户端 IDL 和服务端 IDL 来描述的，这两个 IDL 文件分别用相应的 IDL 编译器编译生成存根代码，它们与桥代码合在一起编译链接产生桥。根据桥的不同生成方法，可将桥划分为以下 3 种类型：

① 静态桥。在静态桥中，SSI 和 CSI 在编译之前就已经用相应接口定义语言描述好，客户能调用的操作集是固定的，它只能对预先定义的操作进行桥接。一旦系统中客户端或服务端的接口发生变化，就必须重新设计和编译桥代码来实现互操作。

② 请求桥。请求桥可以看作静态桥思想的扩展，它是通过调用桥工厂来自动生成所需要的静态桥。静态桥基于已有的服务端和客户端的接口定义，而桥工厂则是根据服务的接口定义自动产生桥代码和客户接口定义，并将桥代码与已经编译好的 CSI 和 SSI 的 Stub 代码连接起来从而实现最终可执行的桥。图 6-9 是请求桥的示意图。

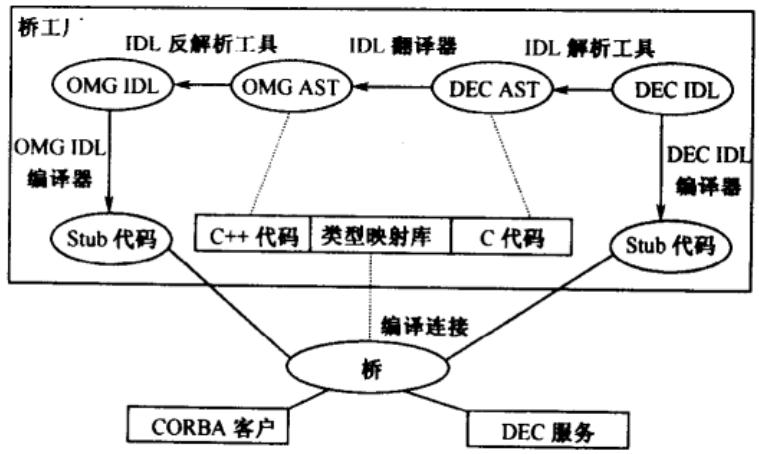


图 6-9　CORBA 客户到 DCE 服务的请求桥

③ 动态桥。动态桥在编译时不需要知道客户和服务的接口类型，它能够支持任意类型的 IDL 接口。实现这种桥，要求互操作的两个平台都具有动态调用接口机制。动态桥具有更大的灵活性，它无需根据每个新的接口定义语言重新构造桥代码，然而 DCE 没有动态调用机制，所以动态桥的实现难度非常大。

在以上 3 种桥中，静态桥适合于接口定义固定不变的环境，在已成型的应用系统中使用是可以的。请求桥使用桥工厂自动产生所需要的桥代码，而不必客户应用程序进行干预，大大减少了开发工作量和程序复杂度，是实现互操作的一种可行方法。而动态桥则尽管有最大的灵活性，且不需要重编译，然而实现过程却过于复杂。所以下面对桥的实现细节的描述只针对请求桥。

3. 用桥实现互操作

用桥技术来实现 CORBA 和 DCE 之间的互操作必须解决好两个主要问题，即接口类型的转换和对象的绑定。

(1) DCE 和 CORBA 接口类型的转换和替代

CORBA IDL 是基于C++ 语法规则的面向对象的客户/服务系统使用的接口语言；而 DCE IDL 则是基于 C 语言规范的面向过程的分布式系统使用的接口语言。它们包括 3 种类型的转换：

① 直接转换。在 CORBA IDL 和 DCE IDL 中有些数据类型具有相同的定义，它们是完全等价的，如一些基本数据类型，以及由这些基本数据类型构成的数组、结构和联合等，可以直接进行转换。

② 间接转换。有些 DCE IDL 中的类型在 CORBA IDL 中没有直接的等价类型，但可以用 CORBA IDL 中的其他类型来进行间接的转换，这包括 DCE 的 integer 类型、指针类型、变长数组、结构和联合等。

③ 不能转换。在两种 IDL 中都有一些类型是各自特有的，如 CORBA 的 context 和 DCE 的管道（pipe）类型等，对于这些类型可以采用一些定制的类型映射来完成类型转换。

(2) 处于两种环境中的对象的绑定

通过 DCE 与 CORBA 的名字服务，利用联合名字和定位器，桥可以将 CORBA 客户端发出的调用请求正确地绑定到相应的 DCE 服务端接口。联合名字由 3 部分组成：名字系统类型、引用环境和名字。定位器根据名字系统类型，选择合适的名字服务确定名字。按照这个过程，在桥的实现中，客户通过联合名字向桥发出服务请求，桥使用定位器得到适合于服务端的名字。要解决名字和地址的一致性问题，还需要扩展绑定概念，并需高层的绑定函数支持联合名字。如果指定对象在使用相同的名字系统的同一个名字空间中，则使用本地绑定函数，否则绑定函数选择合适的桥，通过联合名字与桥连接。

桥工厂在将 CORBA 的 IDL 转换成对等的 DCE 的 IDL 后，就可以根据 CORBA 客户端的接口定义、DCE 服务端的接口定义以及它们之间的映射规则生成桥代码。

桥的思想和设计方法同样适合于其他平台间互操作的实现。

6.4 CORBA 与 EJB 的互操作和集成

CORBA 是一种异构机制之间的互操作规范，它与 EJB 在技术实现上有一定差别，但 EJB 在发布时充分考虑了它与 CORBA 的兼容性。所以，在基于实现的集成中，EJB 比较好的兼容了 CORBA。可以说，EJB 的许多方面在构建时是基于 CORBA 的，它是对 CORBA 规范在互操作实现方面的补充。

从应用系统的集成性角度来看，CORBA 和 EJB 都能很好地实现跨平台、跨网络通信和跨公共服务构件的特性，而 CORBA 在支持多编程语言方面更有优势；从系统的稳定性和开发性的角度来看，CORBA 与 EJB 都有相对完善的事务处理服务、目录服务和安全性服务，而在软件厂商的支持度上 EJB 拥有更为强大的配套解决方案。CORBA 和 EJB 的结合可以充分发挥各自的优势。

6.4.1　CORBA 与 EJB 的关系及其映射规范

在 EJB 的规范中，专门有一类定义了 EJB 到 CORBA 的映射关系，用以实现支持 CORBA 的 EJB 服务器。在这个规范中，EJB 到 CORBA 映射主要分成 4 个部分：部署映射、名字服务映射、事务映射和安全映射。其中，部署映射定义了一个企业级 JavaBean 构件和一个 CORBA 对象之间的部署关系，以及 EJB 规范下 JavaRMI 远程接口与 OMG IDL 之间的映射；名字服务映射规定了如何通过 CORBA 的 COS 名字服务来定位 EJB Home 的对象；事务映射定义了 EJB 事务特性到 CORBA 对象事务服务的映射；安全映射定义了 EJB 规范中安全服务到 CORBA 安全服务的映射。通过这些映射标准，可以实现如下两个方面的功能：

① 在网络上，实现了多个基于 CORBA 的 EJB 容器之间的互操作，这包括不同的软件供应商的名字服务、事务服务和安全服务之间信息的共享。

② 通过标准的 CORBA API 接口，CORBA 客户端能很方便地存取和访问配置在基于 CORBA 的 EJB 服务器中的企业级 JavaBean 构件。

虽然 CORBA 定义了一个在异构系统和异构编程语言环境下的实现不同对象机制互操作的体系结构，但没有定义服务器端的构件协调框架和相应的构件模型。而 EJB 通过定义服务器端构件（或 Bean）和它的容器（container）来增强 CORBA，进而获得了与 CORBA 兼容的服务器端构件协调框架和相应的构件模型。一个容器就是 CORBA 中的对象事务监视器 OTM，而 OTM 是一个 ORB 的事务处理监视器。

为了增强不同厂商 EJB 环境的可互操作性，EJB 定义和约定了一个客户视图到 CORBA 的标准映射，该标准映射包括：

① EJB 体系结构的远程（remote）和本地（home）接口到 RMI－IIOP 的映射。映射是一个标记映射，因为每一个远程和本地接口都是一个 RMI－IIOP 接口；

② 事务上下文在 IIOP 上的传播；

③ 安全上下文在 IIOP 上的传播；

④ 可互操作的命名服务。

这个 EJB 到 CORBA 的映射支持不同厂商的 EJB 容器实现之间的互操作，同时支持非 Java 客户通过标准的 CORBA API 来访问服务器端企业级 Bean 的应用程序。在标准 CORBA 对象服务协议的支撑下，EJB 到 CORBA 映射来传播事务和安全上下文。

6.4.2　CORBA 与 Java 的交互过程描述

图 6-10 直观地展示了 CORBA/Java 的分布式应用模型，其运行过程如下：

① 在 Web 浏览器中，用户向 Web 服务器发送下载某一网页文件的请求；

② Web 服务器向浏览器发送含有 JavaApplet 的网页文件；

③ Web 浏览器载入 Applet，Applet 必须先通过 Java 安全检查后，然后再进入客户机内存；

④ Applet 运行时根据用户的需要通过 Java ORB 使用 IIOP 向服务器端对象部分发送请求；

⑤ 服务器端执行请求，并将结果通过相同途径回传给 Applet；

⑥ Applet 接收结果并在图形用户界面中显示。

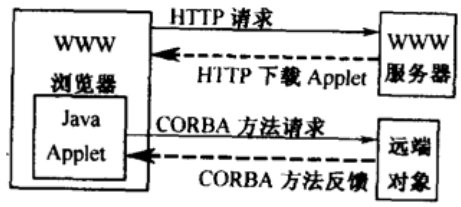


图 6-10 CORBA/Java 分布式实现模型

由于客户端仅需要一个支持 JavaApplet 的浏览器，因此基于 CORBA/Java 的分布式应用系统具有较好的开放性，服务器和客户端任务分布及负载相对比较均衡。客户端下载实现 JavaApplet，因此整个系统的开发较为方便。由此可见，在 Internet/Intranet 广泛使用的今天，基于 CORBA/Java 的分布式应用系统可以在一定程度上解决企业更为复杂的需求。

但是，由于 Java 语言自身的一些特点，如实时性差、JavaApplet 的下载速度较慢等，并不是所有的分布式应用系统都可以采用这种模式来实现。一些对实时性要求较高的系统则不宜采用这种方法实现。

6.4.3 CORBA 结合 EJB 构建分布式对象系统

在 CORBA 与 EJB 结合构建分布式对象系统时，应包括客户端结构和对象服务构件模型两大类。客户端结构是指 EJB/CORBA 型客户端和纯 CORBA 型客户端。对象服务构件模型指标准 CORBA 服务器对象模型和支持 CORBA 的 EJB 服务器构件模型。组织这些构件打造分布式对象系统，这种分布式对象系统具有用户层、中间逻辑服务层和后台数据服务层 3 个层次的结构（图 6-11）。

在这 3 层结构中，用户层交互系统可以使用传统的 Web 浏览器，客户端的请求通过 Java 语言编译成 Servlet 实现 IDL 或者 EJB 接口方式实现；也可根据用户需求定制满意的用户交互系统，建立基于各种应用平台和开发语言的 CORBA 客户端程序。中间逻辑层分布式对象框架是在 CORBA 服务器对象和基于 CORBA 的 EJB 服务器模型的基础上建立的，分别利用 CORBA 构件服务和 EJB 服务

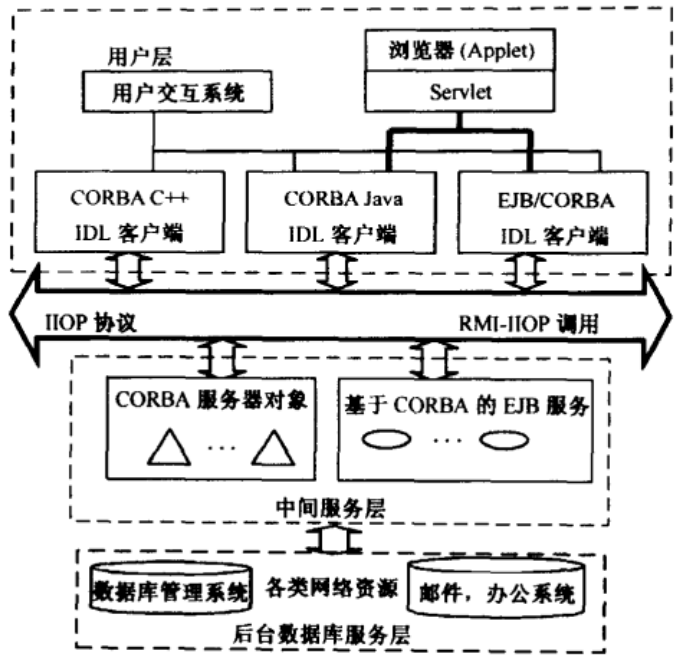


图 6-11　3 层分布结构

实现各自构件对象管理和协调的功能。作为 EJB 服务器端兼容 CORBA 的实现技术，EJB 客户端使用 RMI-IIOP 协议，来完成对 EJB 对象接口的通信。需要注意的是，如果在此结构中 CORBA 服务器对象不是基于 EJB 结构建立的，则不能提供 EJB 类型的客户端对所有 CORBA 服务器对象的访问。而对于各种 CORBA 类型的客户端，都可以实现对 CORBA 服务器对象和 EJB 服务器的访问。后台数据服务层主要建立包括企业数据库、邮件和办公服务等一系列能够通过中间层进行访问的服务项目。

6.5　CORBA 与 Web 的集成

Web 是在 1990 年左右开始兴起的一套通过 Internet 进行访问和信息查询的标准模式，无论是在运行模式上还是软件体系结构风格上，现已形成了一个完整的体系，其核心包括 HTTP、URL、CGI、HTML、Script 等。在此过程中，Web 经历了从超文本 Web 到交互式 Web，再向对象 Web 发展的一个不断成熟的过程。Web 的应用可以说遍布到了人们生活的各个角落。因此，CORBA 与 Web 的结合有其深远的意义。

6.5.1 Web 体系结构描述

以下根据参考文献（董丽等，2000）中提供的内容，对基于 Java 技术与 CORBA 标准的 Web 体系结构和工作机理给予总结和描述。

1. 基于 Java Applet 的 3 层 Web 计算体系结构

图 6-12 所示的 Web 体系结构完全采用纯 Java Applet 开发。这种体系结构可移植性比较好，但在可操作性、可复用性等其他方面表现出了许多不足之处。

在可移植性方面，由于它能适应几乎所有的硬件平台和操作系统，所以具有很好的可移植性。在互操作性方面，虽然不同平台之间的 Java 应用系统可以使用 Java RMI 进行互操作，但与采用其他技术实现的系统之间缺乏互操作的能力；Java 程序可以通过本机代码接口和其他程序一起执行，但是这种接口各方面的可用性都比较差。在可复用性方面，纯 Java Applet 实现的系统和其他使用 Java 技术的系统之间具有一定的可复用性，但是和使用其他技术实现的系统之间几乎没有可复用性。

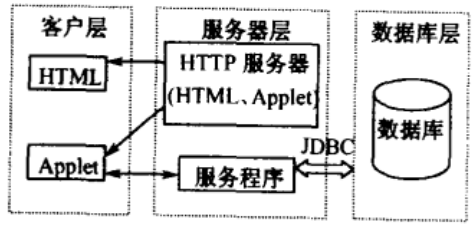


图 6-12 基于 Java Applet 的 3 层 Web 计算体系结构

这种结构在服务器端避免了使用 CGI 程序，而是使用内在支持多线程的Java 程序，在某些方面服务器端的负载有一定程度的减轻。但是，集中式的服务程序，在扩展方面仍然成为系统性能的瓶颈。当连接的用户数增加以致超过服务器能力的时候，除了升级硬件平台以外没有其他可行的方法。此外，在稳定可靠方面，集中式的服务不能得到保证，一旦运行出现故障，很容易导致整个系统的崩溃。系统的客户端采用大量的 Java Applet，造成客户端的网络传输负载很重。另外，这种结构的系统缺乏和 Web 服务器交互的能力，缺乏从服务器端激活客户端的功能。

2. 基于 CORBA/IIOP 的 Web 计算体系结构

图 6-13 是一种基于 CORBA/IIOP（Internet Inter-ORB Protocol）的 Web 计算体系结构。OMG 建立的 OMA，为包括 CORBA 在内的所有规范提供了作为基

础的概念框架。遵从 OMG 的标准，可以开发跨越主要硬件平台和操作系统的应用。

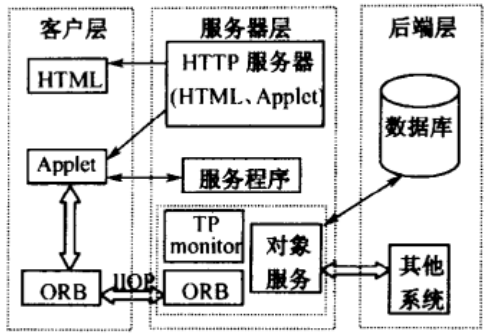


图 6-13　基于 CORBA/IIOP 的 Web 计算体系结构

CORBA 是通用的对象标准，不依赖于各种平台和实现技术，所以基于 CORBA 的系统具有可移植性。开发人员采用 CORBA 规范中通用的接口定义语言 IDL，用标准的方式定义对象的接口，做到了对语言的透明。只要应用系统的实现所使用的语言本身具有可移植性，进而会增强基于 CORBA 的应用系统的可移植性。

在 CORBA 结构中，对象之间的交互是通过 ORB 进行的。采用不同实现技术的对象之间不需要知道对方的实现语言或物理地址，只需要遵从对象接口的定义，发出请求就可以实现在 ORB 内部的互操作。采用这种典型的 CORBA/IIOP 的 Web 计算体系结构，可以保证系统的可移植性、互操作性和可复用性，以及高度的可扩展性。但是这种结构最主要的问题就是客户端的网络传输负载很重(下载大量 Applet 以及 IIOP 通信的开销)，缺乏真正的可用性，因而并不能完全满足 Web 计算环境的需要。

3．基于 Java Servlet 的 4 层 Web 计算体系结构

基于 Java Servlet 的 4 层 Web 计算体系结构如图 6-14 所示。Java Servlet 是运行在面向请求/响应结构的服务器上的模块。它可以运行在支持 Java 的 Web 服务器上，为其提供扩展功能。由于 Java Servlet 支持多线程，开销小，性能高，而且运行安全可靠，具有一致的开发接口，能够方便地与其他系统集成，所以在性能方面优于 CGI。Java Servlet 的出现，为 Java 计算的体系结构提供了对前端 Web 服务的完善支持。

前端服务层能够减少 Applet 下载的时间，还可以减少客户端执行一些例行服务所需要的时间。但是这种基于 Java Servlet 的 4 层计算体系结构如果仅仅采用 Java 技术，而不结合其他技术的话，与 3 层 Java 计算体系结构一样，在可扩

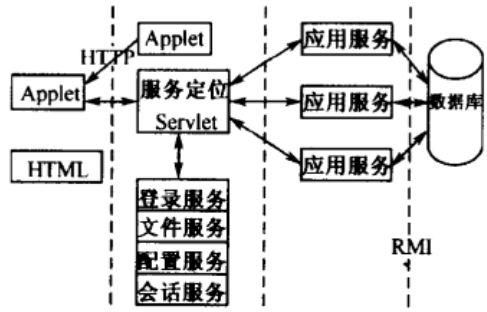


图 6-14 基于 Java Servlet 的 4 层 Web 计算体系结构

展性、容错性、与其他系统的互操作性方面性能较差。为了真正使 Web 计算体系结构适应现代企业的需求，必须综合采用多种技术。

4．基于 CORBA 与 Java Servlet 结合的 Web 计算体系结构

图 6-15 是采用 Java Servlet 与 CORBA 结合的 Web 计算体系结构。在这种体系结构中保留了以前的数据库层和信息搜索引擎部分，而着重是对中间服务层进行了重新调整，将其划分为前端服务层和应用服务层。前端服务的 Java Servlet 分为 3 个模块，分别提供数据库查询、用户管理和信息订阅。应用服务层使用了 JDK 提供的 ORB。由此可见，这种体系结构具有良好的对象化特性，在对象化设计中，使得系统的各个部分之间的大粒度的软件复用性大大提高。

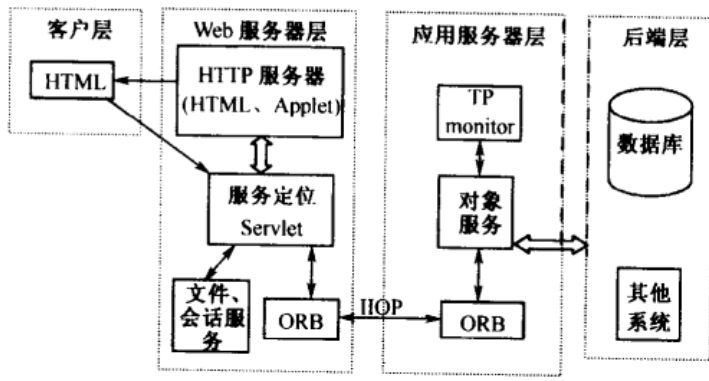


图 6-15 基于 Java Servlet 和 CORBA 的 Web 计算体系结构

通过对几种体系结构的比较可以看出，通用的标准使系统集成成为可能。为了适应将来全球企业计算发展的趋势——面向 Web 的企业计算，在部署企业应用时，必须采取开放的、可扩展性和互操作性好的软件体系结构。为了达到这个要求，体系结构通用标准的建立和选取就显得更为重要。

6.5.2　CORBA 与 Web 的互操作分类

实现 CORBA 与 Web 的互操作和集成大约有 3 种方法：

① 利用传统的 CGI，即将其作为 CORBA 的客户端，利用 CGI 去调用 CORBA对象，如图 6-16 所示。

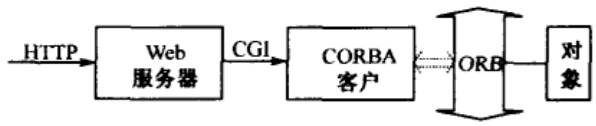


图 6-16　建立在 CGI 之上的互操作模型

②在 Web 的客户端不通过 HTTP 协议，而直接使用 IIOP 去访问 CORBA 对象，或者可以通过增加一个 HTTP/IIOP 转化器，转发对象请求进而完成对 CORBA 对象的访问，如图 6-17 所示。

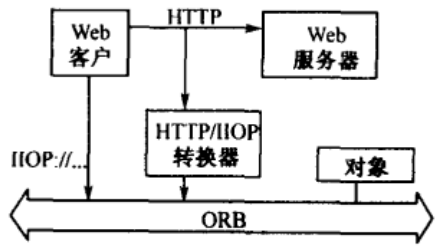


图 6-17　扩充访问协议的互操作模型

③利用 Agent 技术和 Java 的移动代码相结合的技术。这种方式的特点是利用 Java 制作的 Applet，使之工作于客户端，通过 IIOP 直接访问 CORBA 对象，如图 6-18 所示。

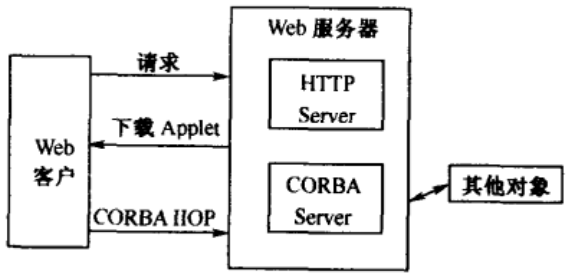


图 6-18　建立在移动代码之上的互操作模型

6.5.3 CORBA 与 Web 集成工作机理

Java 是一种适合于 Internet 面向对象的编程语言，用它开发的程序可以嵌入到 HTML 文档之中，并通过 Internet 从一台计算机传送到另外一台计算机上（客户端）执行。基于 Java 实现的 CORBA 与 Web 结合的 Web 应用体系结构的工作机理如图 6-19 所示。其工作过程如下：

① 客户端用户通过浏览器向服务器发出请求，并且请求的网页内含有 Applet 程序，Applet 由 CORBA 客户程序的 Java 实现，将它与 CORBA 的 ORB 捆绑。

② 浏览器下载并运行 Applet，Applet 创建一个 ORB 服务对象和一组应用对象；之后，Applet 通过 IIOP 访问服务器对象，并由服务器对象完成相应的操作。

③ 通过 IIOP 将请求结果返回给浏览器。

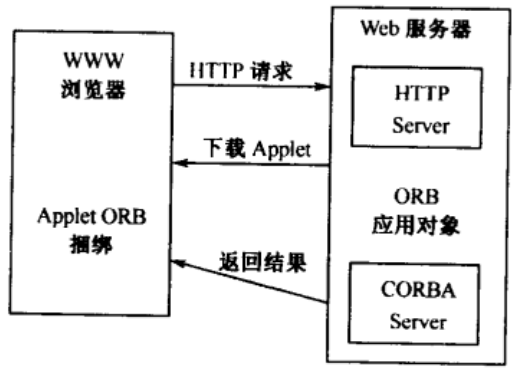


图 6-19 Java/CORBA 的 Web 工作机理

6.6 CORBA 的分布式动态模型

分布式对象的思想是，在分布式系统中引入一种分布的、可互操作的对象机制，但是现在的分布式对象模型（CORBA、DCOM）中的对象是静态的，不利于动态多变的分布式环境。引入移动 Agent 可作为对象模型的扩展，即在对象模型的静态机制中加入了动态的机制，使其适应于更复杂的分布式环境。

近几年来，绝大多数 Agent 平台的研究开发出现了两种趋势，一是基于解释性语言，如 Java 等；二是把移动 Agent 与分布式对象中间件结合起来。在此仅对第二种方式加以描述。

图 6-20 是基于 CORBA 与移动 Agent 相结合的分布式动态模型。其中 Agency 是 Agent 实际运行的环境，每个 Agency 由一到多个能提供特定资源（如服务、

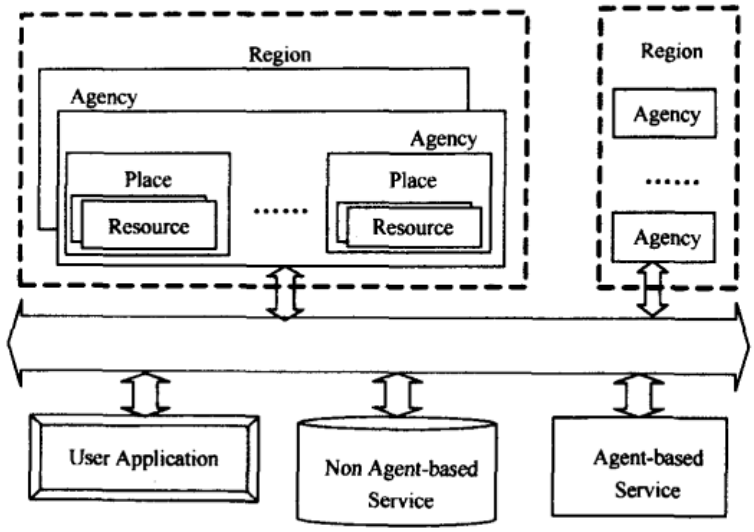


图 6-20　基于 CORBA 的动态分布式平台

内存、文件系统等）的 Place 组成。一个 Region 又由一组 Agency 构成，提供统一的管理策略，以支持管理操作等功能。Agent 的传输、Agent 与 Agent 之间、以及 Agent 与非 Agent 成分之间的交互可通过 CORBA 进行。采用这种集成方式，进一步增强了 CORBA 动态服务能力。

6.7　基于 CORBA 的共享工作空间

6.7.1　共享工作空间的分类及其描述

共享工作空间的含义首先是多个用户共享数据，其次是多个用户对共享数据的并发地、无冲突地访问与修改。共享的实现方式一般有集中式、复制式和分布式 3 种。

（1）集中式

系统中存在着中心服务器。每个工作站缓存了数据，但不能修改数据直到它取得数据的控制权。取得数据的控制权意味着工作站要向中心服务器请求对数据的封锁。在取得数据的控制权后，再由它或中心服务器广播其对数据的修改，让其他工作站各自修改缓存了的数据。

这种方式的优点是，通过中心服务器的控制，可以保证各用户对数据的操作是顺序执行的，各工作站上的数据永远保持一致，是一种比较容易实现的控制机制。

这种方式的缺点是，在系统有多个用户或用户频繁操作数据时会有缺陷。集

中式的 C/S 结构中的服务器端将会是一个瓶颈，因为服务器无法服务于这么多的请求。同样，在一个操作系统中进程数也不能太多，这样，在系统有多个用户或频繁操作数据时，即使在局域网，集中式的 C/S 结构系统的响应时间也会很长。

(2) 复制式

系统中不存在中心服务器。每个工作站各自有一份数据的拷贝。每次对数据进行修改时，都以可逆的方式进行，然后广播其对数据的修改，让其他工作站各自修改自身的数据拷贝。这样可能造成各工作站数据的不一致，因此在广播时要加入时间戳技术，方便各工作站检测有无冲突。冲突发生时采用 Undo/Redo 技术自动纠正或报告用户。

这种方式在广域网中是一种很有效的方式。虽然在这种方式中，对冲突的检测和修改的实现都比较复杂，但因为在广域网中，封锁的申请和释放都要通过远端服务器，网络延时较长，而复制式就能极大地缩短系统响应时间。

复制式系统的缺点是，会增加冲突的概率，这样冲突的纠正算法将会极其复杂，以致于要消耗大量的处理机时间或根本无法实现。

(3) 分布式

在该系统模型中，把共享工作空间中的数据进行分类，每类数据作为一个对象的类。而在系统中的每个数据是一个对象。对象可以分布在系统中的任一台机器上。每个对象都可以接收用户的请求，并发送相应的消息。在这个模型中，每个对象都相当于一个服务器。与传统的集中式 C/S 结构不同的是系统中存在多个服务器。在系统中各用户的请求被分门别类地分发到相应的对象中，而对象是分布在系统的各处，这样，有效地减轻了某一台机器的负荷，均衡了系统的负载。而对于共享的每一个数据来说，仍是采用集中式的控制，因此可以保证各用户对数据操作是顺序执行的，各工作站上的数据永远保持一致。

由此可见，分布式对象结构将集中式和复制式的优点进行了很好的结合，在实际应用中是一种比较理想的选择。

6.7.2 共享工作空间模型的组成

图 6-21 是多对象组成的共享工作空间模型。

在该模型中，客户对象作为客户程序与系统其他对象之间交互的代理。这样，系统可以都用 CORBA 对象来实现，在实际上与逻辑上更流畅。客户程序不再与系统中的其他对象发生关系，而只需同客户对象进行交互，实现起来更灵活。

域对象在逻辑上是控制对象的管理者，一个域对象可以管理多个控制对象。客户加入共享工作空间时，首先向域对象提出申请，指明要加入哪一个共享工作

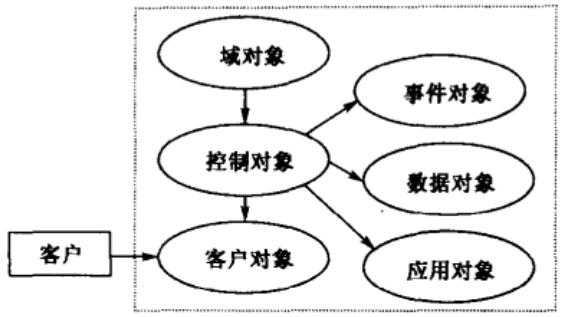


图 6-21　共享工作空间模型

空间。域对象要负责控制对象的创建与删除。

控制对象又称为总控对象，起管理的作用。除了应有常见的管理功能外，在这里，控制对象还负责数据对象的生成与消灭。我们把共享工作空间的控制与管理都笼统地归入控制对象，它除了主要的对象管理功能外，控制对象的功能还包括会话管理、成员管理、对象管理、安全管理、角色管理等功能，这些根据 CORBA 中对象划分的思想，都可以由单个 CORBA 对象来实现，而这些对象可以进一步细分，如安全管理对象要用到认证对象和授权对象的服务等。在共享工作空间中为了方便，一般只有一个控制对象。

数据对象即共享工作空间的共享对象，如白板上的文本、图形、图像、文档、报表等。

在系统中可采用事件激发机制，即一个对象可以创建一个 COSS（公共对象服务规范）事件的通道，并把它放入 COSS 的名字服务中去。其他应用可以通过名字服务取得这个事件对象的引用，并使自己成为这个事件的生产者与消费者。

应用对象又称应用辅助对象，其包括邮件对象、数据库对象和文件对象等，在共享工作空间辅助完成其他的功能。

共享工作空间不是孤立的，对象集成为同其他系统与应用发生关系提供了途径。在这个模型中有多个服务器，也可以说这个模型中没有服务器和客户机的概念，任一个对象可以向其他对象请求服务，也可以向其他对象提供服务，它们之间的地位是平等的。

与 DCOM/CORBA 的结合，可以扩展共享工作空间的应用开发范围。

6.7.3　基于 COM/CORBA 的共享工作空间模型

在 CORBA 标准中，IDL 编译器只能将接口描述映射为 C、C++ 和 Java 等有限的几种语言。这样只能用上述语言进行客户方的应用开发，从而限制了 CORBA 应用的广泛性，形成了开发 CORBA 平台上应用的瓶颈。可以通过 CORBA

与COM的集成（有关COM与CORBA的融合见以上章节的内容）来实现客户端开发应用的广泛性，即通过COM中间件，利用COM支持的众多语言工具，来开发和完成基于CORBA平台的分布式应用。

从COM的观点来看，对象是应用程序的一部分，代表一个接口，从而使这部分应用程序对其他部分或其他应用程序来说是可见的。从CORBA的观点来看，对象是一个能够完成一系列行为的独立成员，在理论上，任何CORBA对象的客户都能透明地访问该对象，并且与对象或客户的位置和实现无关。这一切为基于CORBA/COM融合的多对象共享工作空间模型的集成提供了有效的途径。

传统模式的C/S体系即所谓的胖客户器/瘦服务器，由于客户端承担的任务繁重而无法实现资源的共享，从而产生了目前所倡导的瘦客户器/胖服务器模式，许多大型的软件或数据库开发商也都针对这种变化推出了相应的商用系统。应该承认，瘦客户器/胖服务器以其卓越的性能代表了未来的计算模式。然而，应用于目前的Internet环境，网络通信的响应很大程度上取决于带宽这样的硬件设施以及网络稳定性等一些不确定因素的影响。在目前这种情况下，如果一味地把所有的操作与配置加于服务端，当批量用户同时访问时，服务器对于网络频繁请求的响应可能会异常缓慢。因此，针对网络实际通信状况，在客户端与服务器端实现任务的合理配置就至关重要。也就是说，基于Internet下的B/S体系中如何合理的分配前后端的负载将直接影响系统的响应效率。

建立在CORBA/COM与多对象共享相融合基础上的工作空间模型（图6-22）是一种Internet工作空间共享分布互操作结构模型。

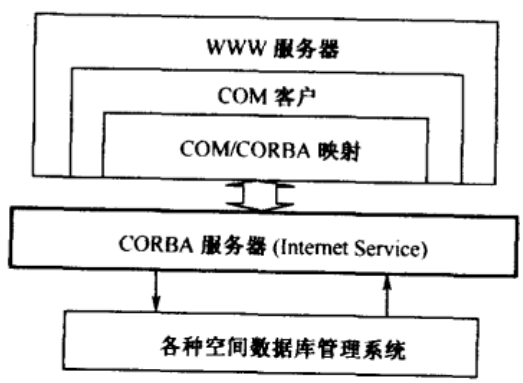


图 6-22 基于COM/CORBA的共享工作空间模型

在该工作空间中，由于无需区分客户端与服务器，因而易于负载均衡的实现。该模型主要是以COM客户作为请求源，以CORBA服务作为目标。对于Internet共享工作空间，实现异构数据源应用级互操作以及分布式数据管理与处理，把现实世界抽象为可互操作的对象是其主要的目标之一。这样，就必须解决

基于 Internet 的分布应用级构件化以及数据的分布式存储和获取。所以，运行于 Internet 的共享工作空间应是一个多客户浏览器/多服务器系统，通过 Internet 的触角将相对独立的部件用网络连接并实现网络范围内的处理。

参 考 文 献

白英彩.2001. 计算机集成制造系统.北京:清华大学出版社

车文富.2001.CORBA 与 OLE/COM 互操作实现技术.计算机工程与应用,13(7):104～106

董丽,李京华,王克宏.2000. 基于 CORBA 的 Web 计算体系结构的研究.清华大学学报,40(9):82～85

韩秋凤,肖政宏.2001. 基于 COM+的数据库访问构件.电脑与信息技术,4(4):36～38

黄庆荣,傅清祥.2001. 基于 CORBA/COM 互操作的 GIS. 福州大学学报(自然科学版),29(4):53～56

来欣等.1999.CORBA 与 OLE/COM 的集成研究.计算机工程与应用,2(2):40～43

戎伟,王汝传等.2000. 开放分布式应用程序的研究.南京邮电学院学报,20(4):62～65

杨晓东等.1999. 基于 CORBA 的 CSCW 共享工作空间的设计与实现.计算机应用研究,11(11):39～41

曾飞鹏,张萌,张启瑞.1999.CORBA 和 DCE 互操作技术.计算机研究与发展,36(3):325～330

郑波,李松年,张世永.2001. 基于 CORBA 的 EJB 体系结构分析.计算机工程与应用,13(13):76～78

Shirley J, Hu Wei, Magid D. 1994. Guide to Writing DCE Applications. Sebastopol, CA: O'Reilly & Associates Inc.

SUN Microsystem Inc. 1998. Enterprise JavaBean Specification. v1.1

第 7 章　数字城市的软件构架模型

信息化、网络化、智能化和可视化是对数字城市的最基本的要求，而数字城市的最终目的是对大量分布在不同地域中的信息进行合理的组织、加工和利用。这是因为：

一是网络化意味着信息的分布性和多源性。在目前网络带宽有限（将来也是，除非实现真正的“无限”网络带宽）的情况下，利用尽量“精简”的信息来表达和代替大量的信息在网络中的传递和管理，是决定数字城市走向成功和实用的关键因素；同时，信息化、智能化和可视化要求我们对数字城市建设中的信息要进行组织的标准化。基于元数据（metadata）的数字城市数据组织模型的研究正是该意义下的具体体现。随着数字地球在全球范围内的进一步研究和深入，人们对元数据在网络化和信息化建设中的重要性有了更进一步的认识，它已被公认为数字地球建设中的 6 大关键技术之一。对作为数字地球中必不可少的成员之一的数字城市的建设，更应首先开展基于元数据的数据组织模型的研究。

二是网络化意味着信息资源的分布性，而分布的资源需要相互合作软件的支持以解决公共问题，Agent 技术为此类问题提供了非常自然的建模方法。随着数字地球在全球范围内的进一步研究和深入，人们对 Agent 在网络化和信息化建设中的重要性有了更进一步的认识，它已成为分布式计算的基本模式和解决网络带宽的有效途径之一。Agent 是一个运行于动态环境的、接收另一个实体的委托并为之提供服务的、具有较高自治能力的实体。其能够在目标的驱动下主动采取各种必要的手段和行为，对动态环境的变化做出适当的反应。Agent 的本质是研究如何使一个或多个实体尽可能地不打扰用户，依靠其自身的能力，采取各种可能的方法和技术完成用户所委托的较为复杂或繁琐的任务。多个 Agent 组成一个较为松散且相互协作的联合体则为多 Agent 系统（MAS），它的主要任务是通过多个 Agent 之间的相互协同和相互服务来共同完成一个任务。

由此可见，数字城市的打造归根到底可由两部分内容组成，即分布式数据的组织和分布式智能软件的构架。本章对这两种模型进行了较深入的讨论，并给出了具体的解决方案。

7.1　基于元数据的数字城市数据组织模型

7.1.1　数字城市中元数据的内涵

（1）元数据的定义

虽然对元数据的定义目前有不同的观点，但通俗地讲，元数据是“关于数据的数据”，也有人将之称作“超数据”。在 MDC 的 OMI 元数据管理模型中，对元数据定义为“元数据是对数据以及对这些数据进行操作的应用和处理过程的描述信息”。笔者认为后一种定义更为合理，因为它能更具体地从“静态”和“动态”两个方面体现元数据的本质和构造机理。其中静态反映在对数据的描述方面，而动态反映在对数据操作应用和处理过程的描述方面。

（2）与元数据相关的几个概念

① 目录及索引卡片：这是人们日常接触最多的一种元数据，其主要目的用于快速定位和检索。

② 数据字典：是数据库系统研究和应用中进行数据管理的基础，它主要从静态方面对所管理的数据集的数据结构和数据内容给以详尽的描述，主要包括原始的数据。数据字典可以看作一种抽象性不高的元数据。

③ 空间元数据：是关于地理空间数据和相关信息资源的描述性信息。它通过对地理空间数据的内容、质量、条件、位置和其他特征进行描述与说明，帮助和促进人们有效地定位、评价、比较、获取和使用地理相关数据。

④ 空间元数据库：对空间元数据实施存储和管理的数据库。

⑤ 空间元数据仓库：是指支持决策过程的、面向主题的、集成的、随时间而变化的、持久的元数据的集合。元数据仓库的建立更有利于超海量分布式数据的管理和利用。

⑥ 数字城市中的元数据：它描述了数字城市中数据集的内容、质量、分类方式、表示方式、参照方式、管理方式和其他特征，是数字城市中数据和信息共享的核心内容之一。

由此可见，数字地球中的元数据的主要功能是如何快速地通过网络检索元数据，并从它所描述的数字城市信息空间中获取所需信息。

（3）数字城市中元数据的形成描述

在数字城市中，各类数据分归于各个部门中进行管理。因而，元数据的形成也应从各个部门中层层进行，最后形成“金字塔”式的元数据体系结构，其类似于多级索引。在 Internet 的支撑下，这种多级索引不受区域的限制。

（4）数字城市中元数据及元数据库的分类

在数字城市中，对元数据的划分和归类亦可按行政管理的“金字塔”模式进行。在此，将数字城市中的元数据划分为部门级元数据（DMD）、企业级元数据（EMD）、行业级元数据（VMD）和城市级元数据（CMD）。DMD 主要用于对部门单位内部各种数据的描述，通过 DMD 可以对部门内部的各种数据进行全面的定位和检索。EMD 是建立在 DMD 基础之上的元数据，即为元数据的元数据，每个 DMD 在 EMD 中各有一个登记项。VMD 是更高一层的元数据，它建立在 EMD 基础之上。CMD 是数字城市中的顶级元数据。DMD、EDM、VMD 和 CMD 形成了层状塔式结构，并提供了多层次的索引管理。

由于数字城市中的元数据可划分为 DMD、EMD、VMD 和 CMD 4 类。相应地，将数字城市中的元数据库亦归为 4 类：部门级元数据库（DMDDB）、企业级元数据库（EMDDB）、行业级元数据（VMDDB）和城市级元数据库（CMDDB），分别用于存放 DMD、EMD、VMD 和 CMD。DMDDB、EMDDB、VMDDB 和 CMDDB 形成了层状塔式结构，并提供了基于元数据的多层次索引框架模型。

7.1.2 数字城市中元数据的特征

在数字城市中，元数据是对其数据的归纳抽象和表达描述。因而，对它特征的进一步研究，更有利于我们对数字城市数据组织模型的建立。

数字城市中的元数据有如下的特征：

① 抽象性。数字城市中元数据是对所表征的数字城市中的实体的一般性描述，具有比普通数据更加丰富的内涵和广泛的语义。因而，抽象性是数字城市中元数据的本质之一。

② 层次性。建立在数字城市数据基础之上的数字城市元数据，可对其进行更深层次的归纳和抽象，形成高层次的元数据，即元数据的元数据（元元数据）。在这种层次模型中，其深度会随着科学技术水平的发展以及人们认知的丰富而加深。

③ 粒度可变性。元数据的粒度是对其大小的度量。在数字城市中，由于被抽象的对象和范围是变化的，因而，数字城市中的元数据的粒度亦是可变的。

在实际应用中，可将粒度大小不同的元数据综合于一体，以适应不同环境的需要。

采用自上而下的方法，对数字城市中的数据进行多层次的综合和抽象，由此形成一棵粒度树。在面向对象的软件工程中，这种粒度树与类和超类相一致。因而，元数据的粒度树在分布式对象构件开发中有更为重要的意义。

④ 区域极小性/精简性。数字城市中的元数据是对一定范围和一定层次上的实体的数据或元数据的抽象，相对于人们现有的认知水平，数字城市中的元数据

可完全表达或基本代替被抽象的实体数据或元数据，是对“大块”数据或元数据的精简和提炼，因而具有区域极小性。

⑤ 可挖掘性。一般情况下，数字城市元数据的提取和应用是建立在海量数据的基础之上，包含了时间维的概念。也就是说，在数字城市中，通过元数据可形成数字城市数据仓库。这正是对数字城市进行“挖掘”的必由之路。

⑥ 标准性。数字城市中的元数据是在一定的规范和标准下，通过对数字城市中的数据或元数据提炼和抽取的产物，标准性为数字城市中元数据的开放和共享提供了保障。

⑦ 聚类性。对数据共性的表达和抽取是数字城市中元数据形成中不可缺少的环节。数据经过抽象后可得到元数据；反之，元数据通过“连接收集”，可以检索出相应被抽象的数据，其有聚集某一类数据的特性，在此称为聚类性。这种特性是数字城市能基于分布式环境的基础。

⑧ 知识性。元数据是人们对客观实体及其规律的进一步认识，其作为一种知识而存在，而且是一种结构性的知识，是智能处理在数字城市中发挥作用的基础。

⑨ 超连接分布性。简化情况下，元数据可以看作一种索引，其隐藏着对被抽象的数据的“指针”，通过这种指针，可以将分布在不同地区和区域上的数据及元数据连接在一起。这种本质的特性为基于 Web 的数字城市开发提供了保障。

7.1.3　基于元数据的数字城市数据组织模型

数字城市是一个巨系统，数据的海量多样性和地域的分布性是它的本质所在，而 Internet 是它的经脉。为了实现对数字城市中数据的有效管理和利用，可根据元数据的形成过程、特征以及对元数据库的分类，建立如图 7-1 所示的、基于元数据的数字城市数据组织模型。

在数字城市数据组织中，按照元数据标准，首先建立集中式的部门级元数据库（DMDDB）。对于中小型企业，可建立集中式企业级元数据库（EMDDB）；而对于跨地域的企业（如石油企业等），一定要建立分布式的企业级元数据库。行业级元数据库（VMDDB）和城市级元数据库（CMDDB），一定要建立在分布式的基础之上。

由于在元数据管理方面，元数据交换标准的统一进度会比元数据管理标准的统一进度要快，在企业内建立以统一的元数据交换标准集成在一起的分布、自治、异构的元数据仓储（repository），在相当长一段时间内是元数据管理系统建立的主要方式。因而，在图 7-1 中采用了分布自治（主要是指 DMDDB）和异构集成相结合的组织模式，依照现有的元数据交换协议，通过 Internet 实现基于元数据的数字城市中数据的组织和管理。

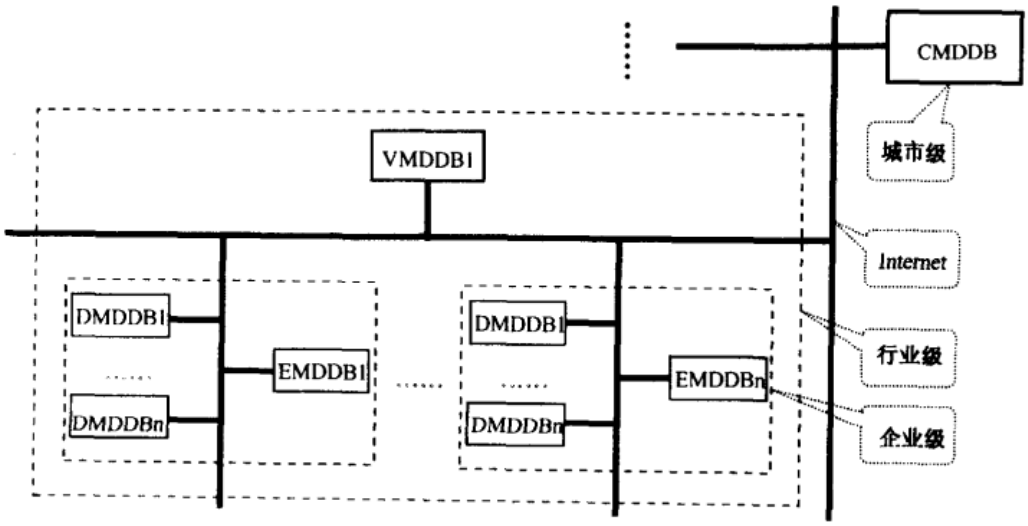


图 7-1 数字城市元数据组织模型

① 部门级元数据库（DMDDB）：部门级元数据库是对本部门数据的总体描述信息的存储体。根据元数据的 ISO/TC211 标准，它包括标识信息、数据质量信息、数据集继承信息、空间数据表示信息、空间参照系信息、应用要素分类信息、发行信息和元数据参考信息等。在图 7-1 所示的模型中，DMDDB 是最底层的元数据库，通过 DMDDB 可直接检索到数字城市中的数据。同时每个 DMDDB 在它所属上一级 EMDDB 中都有一注册项。

② 企业级元数据库（EMDDB）：企业级元数据库对下属企业内部各部门的 DMDDB 和它所在的上属 EMDDB 进行了登记，同时存放了 DMDDB 之间的各种联系。EMDDB 包括了标准信息和引用信息两大部分的内容。而标准信息包括标识信息，如每个 DMDDB 的主机描述符（IP 地址，ODBC 通道、元数据库的位置），以及数据集继承信息、数据表示信息和实体与属性信息等。引用信息包括时间范围信息、联系信息和地址信息等。

③ 行业级元数据库（VMDDB）与 EMDDB 中的内容基本一致，只不过它登记的是其管束的 EMDDB 以及 EMDDB 之间的联系，同时亦记录了它所属的 CMDDB 的相关信息，如主机 IP 地址和相应的访问权限等。

④ 城市级元数据库（CMDDB）：城市级元数据库是数字城市的顶级元数据库，对数字城市中的所有 VMDDB 进行了全面登记。它亦包括标准信息和引用信息两大部分内容，它们的内容与 VMDDB 基本相同，差别之处在于在 CMDDB 中无需记录上一级元数据库的有关信息。通过 CMDDB，可以快速访问到 EMDDB 以及 VMDDB 和 DMDDB，乃至数字城市中的某一具体实体数据。

数字城市元数据模型的研究有重要的应用价值。在信息迅速变化的当今，要有效地发挥计算机网络的优势，及时或实时地发挥各种信息的作用，离不开高效优质的分布式智能可视化软件平台的支撑。基于数字城市元数据组织模型的研究，正是数字城市建设以及软件平台开发中首待解决的关键难题之一。

7.2　基于软件 Agent 的数字城市软件构架模型

7.2.1　软件 Agent 在数字城市中的适应性

数字城市是一个巨系统，而且是一个相当复杂的巨系统，它以错综复杂的各种网络（Internet 网、电信网等）作为信息运载的通道和经络。尤其是随着信息技术的发展和 Internet 的普及，给数字城市注入了新的活力和生机。在数字城市中，信息的种类繁多、分布零散，例如，有城市规划的、交通管理的、商业的、旅游的、文化的、社会的等。Internet 为大家高效充分地获取和共享信息展示了广阔的前景。但是，这些零散的信息，只有依靠智能性的软件才能将数字城市构成一个真正的活生生的有机体。数字城市建造的目的就是将分布在 Internet 上的各种信息，利用可视化技术，迅速准确地展示在人们面前。

在数字城市这个巨系统中，数据的海量多样性和地域的分布性是它的本质所在，而 Internet 是它的经脉。为了实现对数字城市中数据的有效管理和利用，可根据元数据规范，建立基于元数据的数字城市数据组织模型。

从宏观上看，基于 Internet 的数字城市的组织可采用 3 层结构，即信息与资源供给层、中间中介层和用户应用层。数据城市的元数据组织模型为数字城市信息与资源供给层奠定了良好的基础。数字城市的元数据组织模型将对软件 Agent（以下简称 Agent）的应用和发展提供更为自然和广阔的空间。在数字城市这个“信息社会”中，必须体现多用户协同工作模式，即人们工作方式的群体性、交互性、分布性和协作性，这和 Agent 的基本特性不谋而合。

7.2.2　基于 CORBA/DCOM 的软件 Agent 数字城市模型

在数字城市中，各种大小群体都扮演着不同的角色。同时，为了达到自己的目的和完成自己的任务，它们不得不相互合作，从而形成了相互协作的局面。这将为软件 Agent 代替这些角色提供了保障。

1. 数字城市 Agent 的分类

在此，将数字城市中的 Agent 划分为 4 大类（称为基本 Agent）：数据Agent、中介 Agent、可视化 Agent 和应用 Agent。

① 数据 Agent。承担基于元数据模型的数字城市中各类数据的管理、组织和发布任务。

② 中介 Agent。负责传递和过滤来自数据 Agent 发布的数据和信息，并将之提交给相应的应用 Agent。在数据过滤和信息提交过程中，要根据信息所属部门的意图负责信息的安全性。

③ 可视化 Agent。负责数字城市的用户界面，包括用户数据的输入以及信息的可视输出。

④ 应用 Agent。接收来自中介 Agent 的数据和信息，并按照用户的要求、感知环境的变化，通过与其他各类 Agent 的合作，完成特定的任务。

2.4 类基本 Agent 的内部结构

(1) 数据 Agent 的内部结构

数据 Agent 由事件感知模块、数据组织模块、数据提交模块和通信模块 4 部分构成（如图 7-2 所示）。

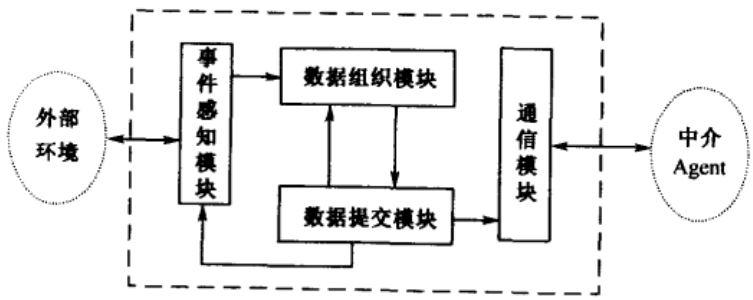


图 7-2 数据 Agent 的内部结构

通信模块负责传递请求和数据交换。事件感知模块接收外部环境的相关信息（在此主要为用户需求信息），并将感知信息交给数据组织模块。数据组织模块将组织好的数据通过数据提交模块进行包装之后，由通信模块传递给中介 Agent。

(2) 中介 Agent 的内部结构

中介 Agent 由事件感知模块、控制模块、安全分发模块和通信模块 4 部分构成（如图 7-3 所示）。

根据外部环境的变化，在控制模块的协调下，由安全分发模块完成信息的安全发派。通信模块负责中介 Agent 分别同数据 Agent 和应用 Agent 的通信任务。

(3) 应用 Agent 的内部结构

应用 Agent 由事件感知模块、通信模块、控制模块、人机界面模块和功能模块 5 部分组成（如图 7-4 所示）。

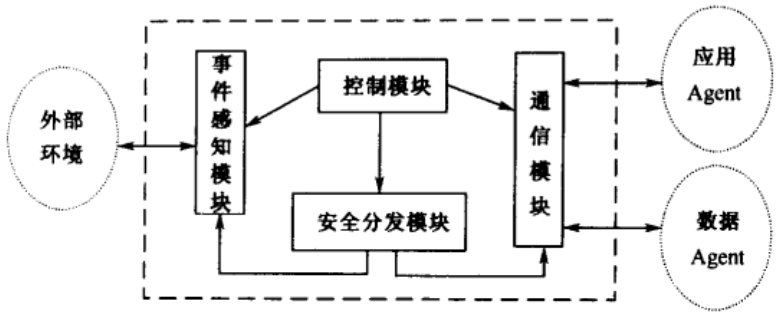


图 7-3　中介 Agent 的内部结构

控制模块是各模块的组织者和管理者，它根据内部数据资源和接收到的消息，进行基于空间信息的推理，并支配功能模块完成相应的任务。人机界面模块是人机交互的接口，它接收用户的请求，完成信息的显示和交互操作等。

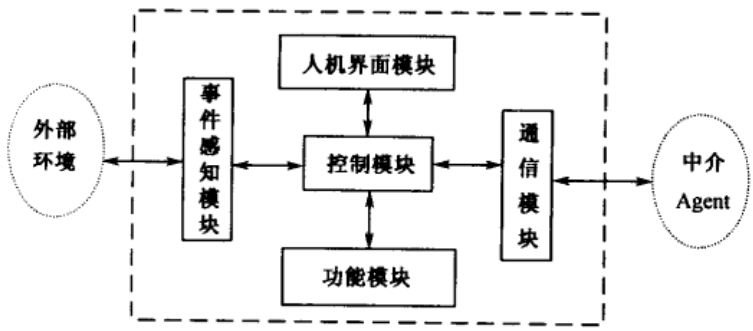


图 7-4　应用 Agent 的内部结构

(4) 可视化 Agent

空间数据分步骤服务模型对数字城市空间数据的显示划分为 4 个处理过程(见图 7-5)。

① 从空间数据源中选择出要显示的地理实体的数据；

② 把选择出来的地理实体数据组合生成一个显示元素的序列；

③ 将显示元素系列生成最终要显示的地图结果；

④ 将准备好地图送往显示设备进行最终显示。

在此，将模型的 4 个步骤分别称作空间数据的选择、显示序列的构造、地图的生成和显示。每一个步骤都接收某一特定形式的空间数据作输入，并输出某种形式的中间结果，每一个上面步骤的顺利执行都要先执行其下相邻的步骤，并用下面步骤提供的输出结果。就是说，上面步骤要调用下面步骤为其服务，下面步骤要为上面步骤提供服务。

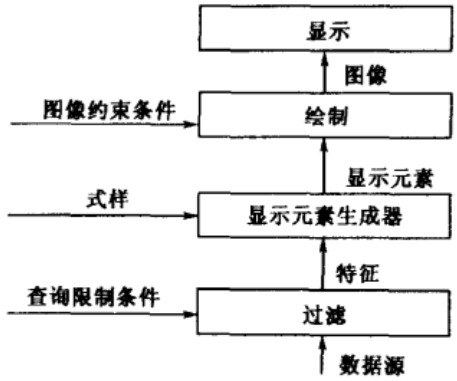


图 7-5 地学空间数据可视化过程

在分步骤服务模型下，将每一步骤的服务用 Agent 加以实现，为实现数字城市的跨平台分布式互操作结构提供了极大的方便。在具体的实现过程中，要处理好各 Agent 的协调和通信问题。采用这种模型的系统，就可以保证每个系统的上面一个步骤的 Agent 可以通过消息传递，通知其他系统的相应下面步骤的 Agent 为之提供服务和完成相应的功能。可视化 Agent 的协作结构如图 7-6 所示。

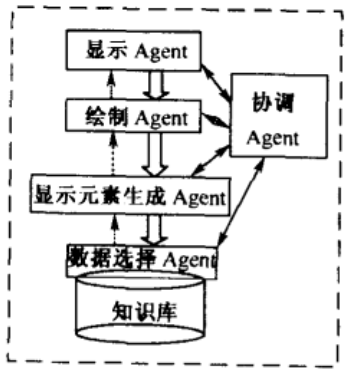


图 7-6 多个 Agent 组成的可视化 Agent

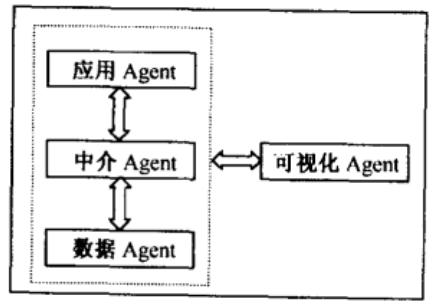


图 7-7 中级 Agent 4 层体系结构

3. 中级 Agent 4 层体系结构

中级 Agent 是由上述 4 个基本 Agent 组成的多 Agent 系统，它为 4 层体系结构（如图 7-7 所示）。中级 Agent 是数字城市中功能齐全的最小 Agent，在基于 Agent 的数字城市组织模型中，将之作为最小的实体单元出现。

中级 Agent 的 4 层结构既有利于基于应用 Agent 的软件的独立性；同时，它为各部门数据和信息的安全共享提供了保障。

4. 基于软件的 Agent 数字城市层状分布模型

在数字城市这个巨系统中，广泛分布的数据和各种各样的信息都具有部门性和行业性。因而，采用与元数据管理模型（图 7-1）相一致的模式来组织和管理数字城市中的 Agent，是一种更为自然和非常有效的方案（如图 7-8 所示）。

在图 7-8 的多 Agent 构成的数字城市模型中，企业级的各类中级 Agent 需通过企业级 AgentServer 来管理。而企业级 AgentServer 和行业级 AgentServer 分别由行业级和顶级 AgentServer 分别进行管理。由此形成了基于多 Agent 的层状组织模型。

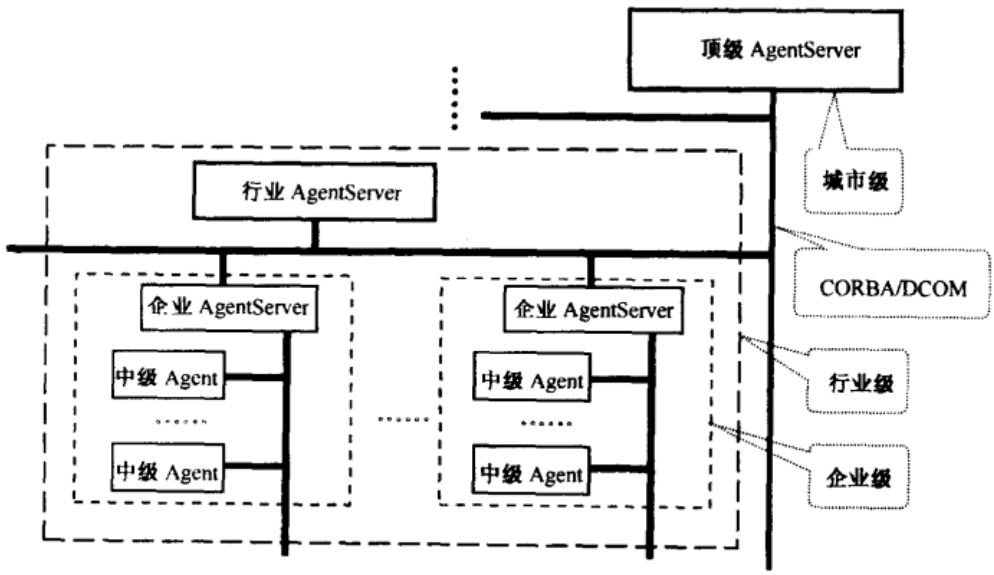


图 7-8　数字城市层状分布模型

建造基于协作关系的数字城市模型，有效地解决了数字城市系统中的分布性、互操作性、协同性和开放跨平台性，它能按照人们处理日常事务的方式进行动态协调地维持数字城市的运转；同时，基本上解决了跨区域的数字城市系统在 Web 上的瓶颈问题。

数字城市是一个全新的概念，如何组织和筹建数字城市？数字城市到底基于什么样的模型才能适应这种巨型系统的安全和顺利运转？被誉为 21 世纪计算机科学领域的一项重要关键技术的 Agent 以什么样的模式在数字城市中发挥它的作用？这一系列的问题人们正处于探讨和研究之中。在数字城市中，要有效地发挥计算机网络的优势，及时或实时地再现各种信息的作用，离不开高效优质的分布式智能软件平台的支撑。本书中提出的基于软件 Agent 的数字城市组织模型无疑

在这方面有一定的开创性。

参考文献

曹蓟光，王中康 2001. 元数据管理策略的比较研究. 计算机应用，38(3)：3～5

陈 军，蒋 捷. 2000. 多维动态 GIS 的空间数据建模、处理与分析. 武汉测绘科技大学学报，25(3)：189～194

陈章渊等译. 1999. 智能 CORBA. 北京：电子工业出版社

程显毅，懂宏斌. 2000. 设计 Agent 系统应注意的问题. 计算机工程与应用，36(11)：64～65

承继成，李琪，易衫桢. 1999. 国家空间信息基础设施与数字地球. 北京：清华大学出版社

戈尔(美国). 1998. 数字地球：二十一世纪认识地球的方式. http://www.creation.com

郭仁忠. 1997. 空间分析. 武汉：武汉测绘科技大学出版社

毋河海，龚健雅. 1998. 地理信息系统(GIS)空间数据结构与处理技术. 北京：测绘出版社

孙艳春. 2001. CSCW 系统体系结构中协作机制的研究. 小型微型计算机系统，22(10)：1182～1185

汪小林，罗英伟，丛升日等. 2001. 空间元数据研究及应用. 计算机研究与发展，38(3)：321～327

王映辉，周明全. 2001. 数字城市关键技术及其模型研究. 计算机工程与应用，1(1)：10～12

王映辉，周明全，耿国华. 2001. 面向软件 Agent 的数字城市组织模型研究. 计算机科学，28(9)：30～32

吴刚，王怀民，吴泉源等. 2001. 一个基于 CORBA 和移动智能体的分布式网管集成框架. 计算机学报，24(1)：19－24

杨世忠. 2000. 实施数字北京的技术思想及方法. http//www.gov.cn

杨树强，王峰，陈火旺. 1997. 面向对象的 3 级数据模型. 软件学报，8(7)：505～510

易文韬，陈颖平等. 2001. JAVA2 程序设计实物务入门. 北京：中国铁道出版社

赵龙文，侯义斌. 2000. Agent 的概念模型及其应用技术，计算机工程与科学，22(6)：75～79

朱淼良，邱瑜. 2001. 移动代理系统综述. 计算机研究与发展，38(3)：16～25

Hyacinth S. Nwana. 2001. Software Agents: An Overview. http://agents.umbc.edu/introduction/ao/